



Electronics Club
IIT KANPUR

Doodlebot

SUMMER PROJECT 2025

Project Duration : May 13th 2025 to July 2025

August 10, 2025

Contents

Section	Page
Project Description	3
System Overview	
User Drawing Input	
Stroke Capture and Processing	
Robot Motion and Drawing Execution	
Hardware	3
Flow-Chart	4
System Architecture Diagram	
Week-0	5
Introduction to Microcontrollers	
Dual Potentiometer and Buzzer Task	
LED Sequence Control	
Computer Vision Path Extraction	
Image Input and Preprocessing	
Median Blur and Thresholding	
Contour Detection and SVG Generation	
PCB Designing in KiCAD	
Week-1	10
PID Controller on SimuLink	
PID Control Theory	
Week-2	10
Chassis Design in Fusion360	
ArUco Marker Detection	
GUI Drawing Application (Tkinter)	
Week-3	15
Camera Module Interfacing with RPi	
DC Motor Control with RPi	



Week-4	16
User Doodle GUI and Logic	
Touch Controller Integration	
3D Printed Parts Assembly	
After Week-4	22
UDP communication	
Problems faced	
Changes In Original Design	
Final Code	
Outcome	21
Sketch Capture and Conversion	
Coordinate Mapping and Motion Control	
Precision Drawing	
Mentors	22
Mentees	22

1 Project Description

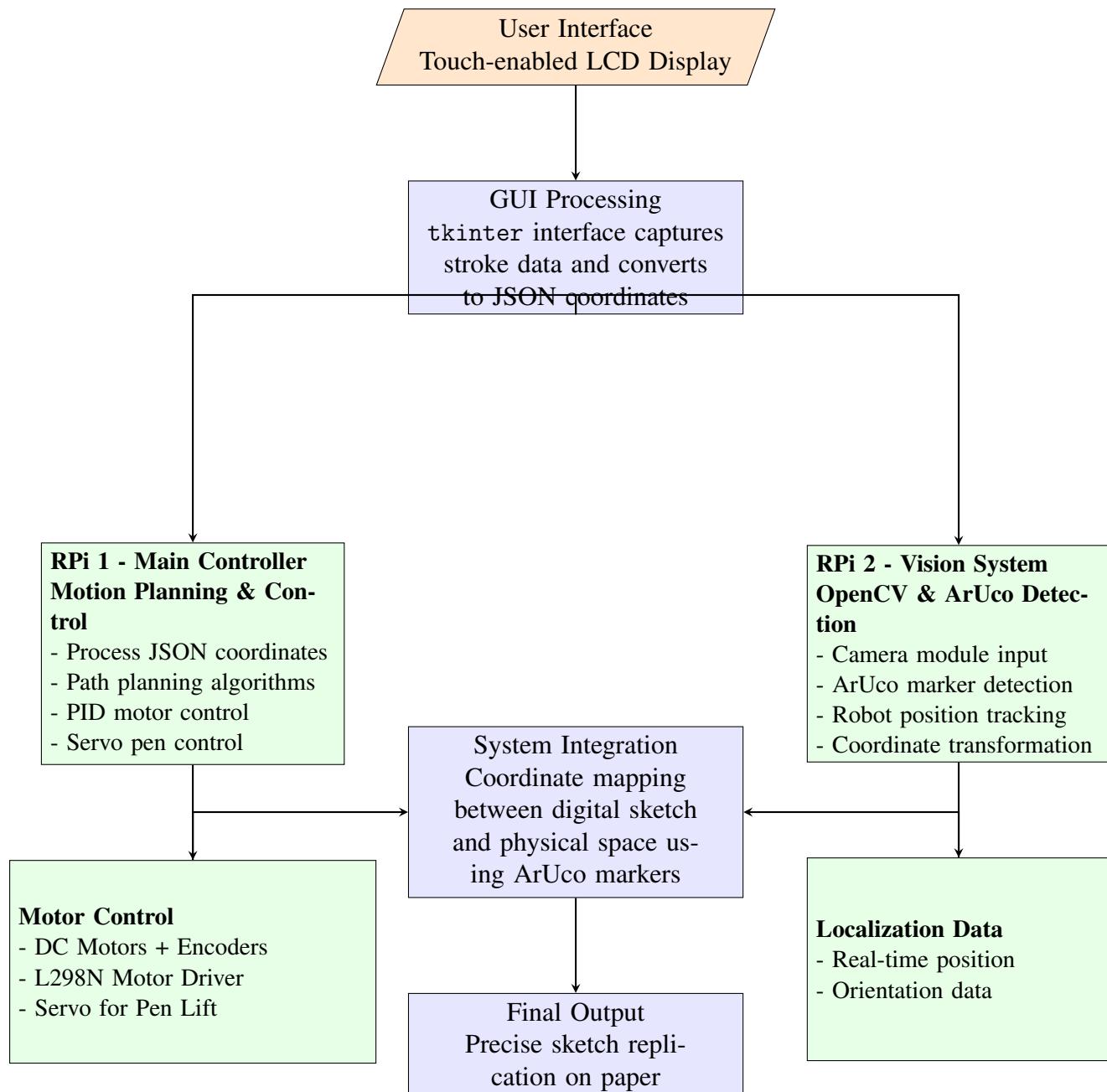
- DoodleBot is an autonomous robot that draws the sketch drawn by the user on a LCD display.
- The user draws on a display, and the robot captures stroke data as an image input.
- OpenCV algorithm captures the strokes in path coordinates. These coordinates are processed for motion planning.
- A Raspberry Pi controls the 2-WD robot with
 - DC motors+encoder for precise movement.
 - A servo motor for the pen lifting mechanism (up/down movement).
- The robot physically replicates the drawing using a mounted pen on the paper, combining display input, image processing, and motor control for a real-time sketch.

2 Hardware

- Raspberry Pi 4
- 2-WD with DC motors+encoder
- Servo motor
- Motor Driver L298N
- Camera module
- touch-enabled LCD display
- ArUco markers
- Castor Wheel



3 Flow-Chart



4 Week-0

- **Introduction to Micro controllers**

- We had an introductory lecture to what a microcontroller is, which is basically a small computer on a single integrated circuit that is designed to control specific tasks within electronic systems. We were given a task of:
 - * 1. Dual Potentiometer Input:
 - Potentiometer 1 (POT1): Controls three buzzers, each corresponding to a range of analog values:
 - Buzzer 1: Activated when analog input is within the first third of the range (0 to 341).
 - Buzzer 2: Activated when input is in the second third (342 to 682). and so on
 - Potentiometer 2 (POT2): Controls the speed at which a sequence of five LEDs lights up one at a time (e.g., 01000, 00100, 00010, 00001, etc. these the lights at any given time). The analog value from POT2 determines the delay between each LED transition.

- **Computer Vision Path Extraction**

- **Image Path Extraction and SVG Conversion using Computer Vision**

- The task was to process a scanned hand-drawn image and convert the prominent inner shape into a scalable vector graphic (SVG) format. This involved basic image pre-processing, boundary detection, and SVG path generation using Python library OpenCV.

```
1 import numpy as np
2 import cv2 as cv
3 import matplotlib.pyplot as plt
4
5 img = 'image_1.jpg'
6 image = cv.imread(img)
7
8 plt.imshow(image)
9 plt.show()
10
11 gray = cv.cvtColor(image, cv.COLOR_RGB2GRAY)
12 blur_med = cv.medianBlur(image,7)
13 A = 20
14 for i in range(A):
```

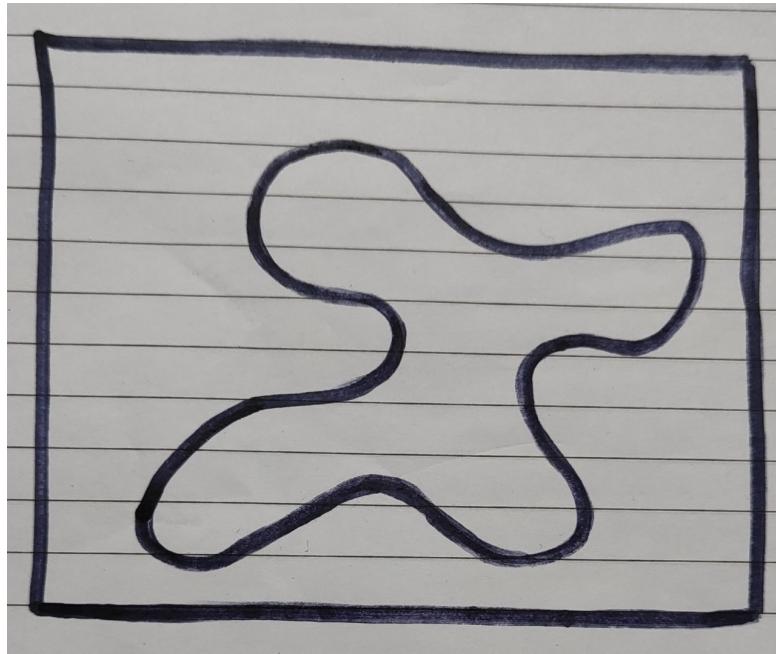


Figure 1: Input image for our OpenCV task

```
15     blur_med = cv.medianBlur(blur_med,7)
16 plt.imshow(blur_med)
17 plt.show()
18
19 gray_1 = cv.cvtColor(blur_med, cv.COLOR_RGB2GRAY)
20
21 ret, th1 = cv.threshold(gray_1, 100, 150, cv.THRESH_BINARY) # change threshold
   values as per the marked path
22
23 plt.imshow(th1, cmap='gray')
24 plt.show()
25
26 plt.ion()
27
28 B = 5 #this loop is to smoothout the curve if there are any rugged lines left
29
30 for i in range(B):
31     th1 = cv.blur(th1,(3,3),cv.BORDER_DEFAULT)
32     ret, th1 = cv.threshold(th1, 100+(2*i), 150-(2*i), cv.THRESH_BINARY)
33
```

```
34     plt.clf()
35     plt.imshow(th1, cmap='gray')
36     plt.title(f'Iteration {i+1}')
37     plt.pause(.5)
38
39 plt.ioff()
40 plt.show()
41
42 N = 135
43
44 cropped = th1[N:-N, N:-N]
45
46 plt.figure(figsize=(10,5))
47 plt.subplot(1,2,1), plt.imshow(cv.cvtColor(th1, cv.COLOR_BGR2RGB)),
48 plt.title("Original")
49
50 plt.subplot(1,2,2), plt.imshow(cv.cvtColor(cropped, cv.COLOR_BGR2RGB)),
51 plt.title("Edges Removed")
52 plt.show()
53
54 contours, hierarchy = cv.findContours(cropped, cv.RETR_CCOMP, cv.
55                                         CHAIN_APPROX_NONE)
56
56 Path = np.zeros_like(cropped)
57
58 for i, h in enumerate(hierarchy[0]):
59     if h[3] == -1:
60         cv.drawContours(Path, contours, i, 255, thickness=1)
61
62 plt.imshow(Path, cmap='gray')
63 plt.axis('off')
64 plt.show()
65
66 selected_contour = None
67
68 for i, h in enumerate(hierarchy[0]):
69     if h[3] == -1:
70         selected_contour = contours[i]
71         break
72
```



```
73 points = [(pt[0][0], pt[0][1]) for pt in selected_contour]
74
75 import csv
76
77 with open('outer_path.csv', 'w', newline='') as f:
78     writer = csv.writer(f)
79     writer.writerow(['x', 'y'])
80     writer.writerows(points)
81
82 with open('outer_path.svg', 'w') as f:
83     f.write('<svg xmlns="http://www.w3.org/2000/svg" version="1.1">\n')
84     f.write('<polyline points="')
85     f.write(' '.join([f'{x},{y}' for x, y in points]))
86     f.write('" style="fill:none;stroke:black;stroke-width:1"/>\n')
87     f.write('</svg>')
```

1) Image Input:

- The hand-drawn image was first read using OpenCV's `cv2.imread()` function.

2) Median Blur:

- To reduce noise while preserving edges, a median blur was applied instead of a normal blur. This was executed multiple times in a loop to improve edge retention and noise suppression.

3) Thresholding:

- Binary thresholding was used to extract the prominent boundaries. This made the inner shape appear distinct from the background.

4) Smoothing the Contour:

- Repeated application of thresholding and blurring helped to smooth the extracted boundary curves. This step ensured the extracted path would be clean and continuous.

5) Boundary Cropping:

- The outer rectangle was manually cropped using parameter `N`, which controlled how many pixels from the edges were removed to avoid detecting the outer frame.



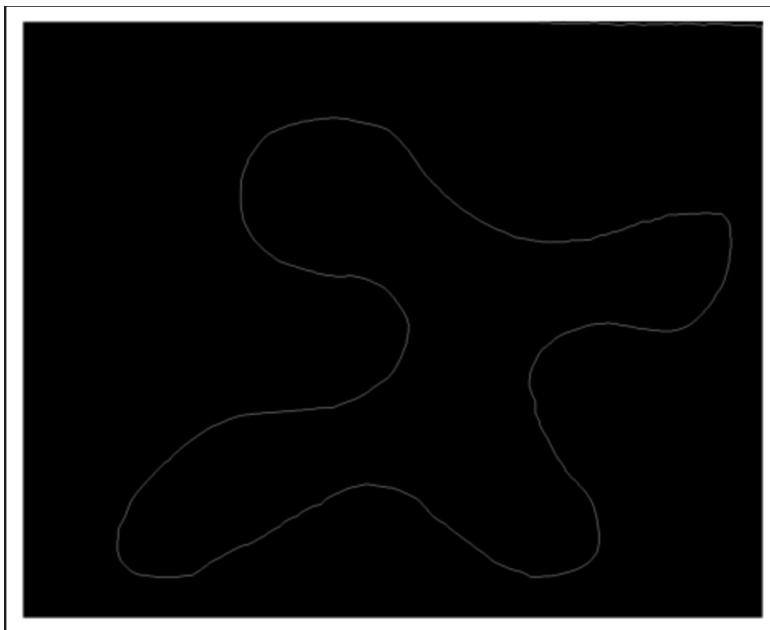


Figure 2: Output of OpenCV task

6) Contour Detection:

- Contours were extracted using OpenCV's `findContours` function. This yielded a list of all curves present in the image.

7) Extracting the Outermost Path:

- Contours with no parent in the hierarchy were identified as outermost. This ensured that the correct path was selected by using the hierarchy information

8) Coordinate Extraction:

- The coordinates of the selected contour were extracted into a list of (x, y) points which were used to generate both CSV and SVG files.

9) SVG Path Generation:

- The final step was to convert the list of coordinates into an SVG path. This was achieved by formatting the points into SVG path syntax and saving them using Python's file handling tools

10) Output:

- The output of the process is a smooth SVG path accurately representing the inner curve of the image. The final file `outer_path.svg` can be used for various vector-based applications like laser cutting or digital rendering.

• PCB designing

- We learned how to design a PCB on KiCAD.



5 Week-1

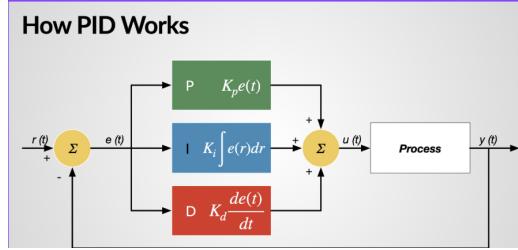


Figure 3: PID controller

- PID controller on SimuLink

- PID (Proportional-Integral-Derivative) control is a feedback control loop mechanism used in various industrial and automated systems to regulate process variables like temperature, flow, pressure, and speed. It uses a combination of proportional, integral, and derivative actions to maintain the system output close to a desired setpoint

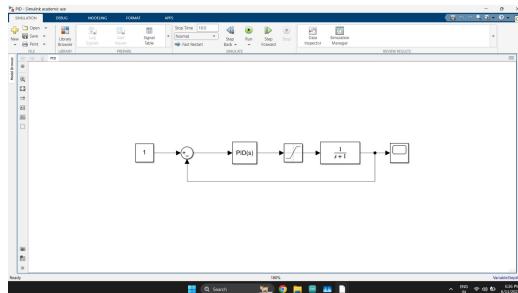


Figure 4: PID controller in Simulink

6 Week-2

- Designing the Chassis using Fusion360

- Fusion 360 being a cloud-based 3D CAD tool by AutoDesl. On Fusion 360, we designed the chassis for 3-D printing.

- ArUco marker

- ArUco markers are black-and-white square fiducial markers used for camera pose estimation and object tracking in computer vision applications. Each marker encodes a unique ID within a grid-like pattern, which allows it to be easily detected and distinguished from others using image processing.

```
1 import cv2
2 import numpy as np
3
4 # Load the ArUco dictionary
5 aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_100)
6 aruco_params = cv2.aruco.DetectorParameters()
7 # Open the camera
8 cap = cv2.VideoCapture(0)
9 while True:
10     # Read a frame from the camera
11     ret, frame = cap.read()
12     if not ret:
13         break
14     # Convert the frame to grayscale
15     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
16     # Detect the ArUco markers
17     corners, ids, rejected = cv2.aruco.detectMarkers(gray, aruco_dict,
18         parameters=aruco_params)
19     # If markers are detected, draw them on the frame
20     if ids is not None:
21         cv2.aruco.drawDetectedMarkers(frame, corners, ids)
22     # Display the frame
23     cv2.imshow('ArUco Marker Detection', frame)
24     # Exit if 'q' is pressed
25     if cv2.waitKey(1) & 0xFF == ord('q'):
26         break
27     # Release the camera and close all windows
28 cap.release()
cv2.destroyAllWindows()
```

1) Marker Detection System Initialization: The system initializes with a predefined ArUco dictionary (DICT_4X4_1000) containing 1000 unique markers, each 4x4 in size. The marker size is set to 18.7 cm for distance calculation purposes.

2) Camera Frame Processing: Each frame from the camera is first converted to grayscale using cv2.cvtColor() since ArUco detection works more efficiently on grayscale images rather than color images.



3) Marker Detection Algorithm: The detectMarkers() function uses OpenCV's ArUco detector to scan the grayscale frame and identify square patterns that match the predefined dictionary markers.

4) Marker Information Extraction: For each detected marker, the system extracts the marker ID, calculates the centroid using image moments, and stores the four corner coordinates for further processing.

5) Distance Calculation: The system calculates the real-world distance to each marker by comparing the pixel size of the detected marker with the known physical size (18.7 cm) using the formula: $\text{distance} = (\text{marker_size_cm} \times \text{frame_width}) / \text{marker_size_pixels}$.

6) Visual Marker Annotation: Detected markers are visually annotated on the frame with their ID numbers, bounding boxes, and calculated distances displayed as text overlays near each marker.

7) Centroid Marking: The centroid of each detected marker is marked with a magenta circle to clearly indicate the center point used for position tracking and distance measurements.

8) Inter-marker Distance Calculation: When multiple markers are detected, the system draws lines between all pairs of markers and calculates the pixel distance between their centroids for spatial relationship analysis.

9) Real-time Display: All processing results are displayed in real-time on the video feed, showing detected markers, distances, connections, and annotations for immediate visual feedback.

10) System Integration: The marker detection system serves as the foundation for robot localization, providing precise position and orientation data that can be used for navigation and coordinate mapping in the DoodleBot project.

7 Week-3

- **Interfacing camera module with RPi**

- The Raspberry Pi Camera Module is a small, lightweight, and high-resolution camera used for image and video capture. It connects directly to the Raspberry Pi board via the CSI (Camera Serial Interface) port. We got familiar by using camera module through RPi.

- **Controlling DC motor with RPi**

- PID (Proportional-Integral-Derivative) control is a widely used feedback mechanism in control systems. It continuously calculates an error value as the difference between a



desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms.

```
1 const int motorPin1 = 24;      // IN1
2 const int motorPin2 = 23;      // IN2
3 const int enablePin = 25;      // ENA (PWM)
4
5 // Encoder pins
6 const int encoderPinA = 17;    // Interrupt pin
7 const int encoderPinB = 27;    // Interrupt pin
8
9 // PID parameters
10 double Kp = 2.0;             // Proportional gain
11 double Ki = 0.1;              // Integral gain
12 double Kd = 0.5;              // Derivative gain
13
14 // Variables
15 volatile long encoderCount = 0;
16 double currentAngle = 0;
17 double targetAngle = 0;
18 double error = 0;
19 double previousError = 0;
20 double integral = 0;
21 double derivative = 0;
22 double output = 0;
23
24 // Timing
25 unsigned long previousTime = 0;
26 const unsigned long sampleTime = 10; // 10ms sample time
27
28 // Motor specifications
29 const int pulsesPerRevolution = 360; // Adjust based on your encoder
30 const double degreesPerPulse = 360.0 / pulsesPerRevolution;
31
32 // Deadband to prevent oscillation
33 const double deadband = 1.0; // degrees
34
35 void setup() {
36     Serial.begin(9600);
37
38     // Motor pins
39     pinMode(motorPin1, OUTPUT);
40     pinMode(motorPin2, OUTPUT);
```



DoodleBot Report

```
41  pinMode(enablePin, OUTPUT);
42
43 // Encoder pins
44 pinMode(encoderPinA, INPUT_PULLUP);
45 pinMode(encoderPinB, INPUT_PULLUP);
46
47 // Attach interrupts
48 attachInterrupt(digitalPinToInterrupt(encoderPinA), encoderISR_A,
49   CHANGE);
50 attachInterrupt(digitalPinToInterrupt(encoderPinB), encoderISR_B,
51   CHANGE);
52
53 Serial.println("DC Motor PID Controller Ready");
54 Serial.println("Enter target angle (degrees): ");
55 }
56
57 void loop() {
58 // Check for serial input
59 if (Serial.available() > 0) {
60   targetAngle = Serial.parseFloat();
61   Serial.print("Target angle set to: ");
62   Serial.print(targetAngle);
63   Serial.println(" degrees");
64
65   // Reset integral term when new target is set
66   integral = 0;
67 }
68
69 // PID control loop
70 unsigned long currentTime = millis();
71 if (currentTime - previousTime >= sampleTime) {
72
73   // Calculate current angle from encoder
74   currentAngle = encoderCount * degreesPerPulse;
75
76   // Calculate error
77   error = targetAngle - currentAngle;
78
79   // Check if within deadband
80   if (abs(error) <= deadband) {
81     stopMotor();
82     if (abs(error) <= deadband && abs(previousError) > deadband) {
83       Serial.print("Target reached! Current angle: ");
```



DoodleBot Report

```
82     Serial.print(currentAngle);
83     Serial.println(" degrees");
84 }
85 } else {
86 // Calculate PID terms
87     double dt = (currentTime - previousTime) / 1000.0; // Convert to
88     seconds
89
90     integral += error * dt;
91     derivative = (error - previousError) / dt;
92
93     // Calculate PID output
94     output = Kp * error + Ki * integral + Kd * derivative;
95
96     // Limit integral windup
97     if (integral > 100) integral = 100;
98     if (integral < -100) integral = -100;
99
100    // Apply output to motor
101    controlMotor(output);
102}
103
104// Update previous values
105previousError = error;
106previousTime = currentTime;
107
108// Print status every 500ms
109static unsigned long lastPrint = 0;
110if (currentTime - lastPrint >= 500) {
111    Serial.print("Target: ");
112    Serial.print(targetAngle);
113    Serial.print(" , Current: ");
114    Serial.print(currentAngle);
115    Serial.print(" , Error: ");
116    Serial.print(error);
117    Serial.print(" , Output: ");
118    Serial.println(output);
119    lastPrint = currentTime;
120}
121}
122
123void controlMotor(double pidOutput) {
```



DoodleBot Report

```
124 // Limit output to motor range
125 pidOutput = constrain(pidOutput, -255, 255);
126
127 if (pidOutput > 0) {
128     // Rotate clockwise
129     digitalWrite(motorPin1, HIGH);
130     digitalWrite(motorPin2, LOW);
131     analogWrite(enablePin, abs(pidOutput));
132 } else if (pidOutput < 0) {
133     // Rotate counter-clockwise
134     digitalWrite(motorPin1, LOW);
135     digitalWrite(motorPin2, HIGH);
136     analogWrite(enablePin, abs(pidOutput));
137 } else {
138     stopMotor();
139 }
140
141
142 void stopMotor() {
143     digitalWrite(motorPin1, LOW);
144     digitalWrite(motorPin2, LOW);
145     analogWrite(enablePin, 0);
146 }
147
148 // Encoder interrupt service routines
149 void encoderISR_A() {
150     if (digitalRead(encoderPinA) == digitalRead(encoderPinB)) {
151         encoderCount++;
152     } else {
153         encoderCount--;
154     }
155 }
156
157 void encoderISR_B() {
158     if (digitalRead(encoderPinA) != digitalRead(encoderPinB)) {
159         encoderCount++;
160     } else {
161         encoderCount--;
162     }
163 }
164
165 // Function to reset encoder position (call this to set current
166 // position as 0)
```



```
166 void resetEncoder() {
167     encoderCount = 0;
168     currentAngle = 0;
169     Serial.println("Encoder position reset to 0 degrees");
170 }
171
172 // Function to tune PID parameters via serial
173 void tunePID() {
174     Serial.println("Enter new PID values:");
175     Serial.print("Kp (current: "); Serial.print(Kp); Serial.print("): ");
176     while(!Serial.available()) {}
177     Kp = Serial.parseFloat();
178
179     Serial.print("Ki (current: "); Serial.print(Ki); Serial.print("): ");
180     while(!Serial.available()) {}
181     Ki = Serial.parseFloat();
182
183     Serial.print("Kd (current: "); Serial.print(Kd); Serial.print("): ");
184     while(!Serial.available()) {}
185     Kd = Serial.parseFloat();
186
187     Serial.println("PID parameters updated!");
188 }
```

8 Week-4

- Created GUI to accept user input doodles

```
1 import tkinter as tk
2 from tkinter import ttk, messagebox, filedialog
3 import json
4
5 class DrawingApp:
6     def __init__(self, root):
7         self.root = root
8         self.root.title("Drawing Interface")
9         self.root.geometry("650x500")
10
11         # Canvas dimensions
12         self.canvas_width = 600
13         self.canvas_height = 400
```



DoodleBot Report

```
14
15     # Drawing variables
16     self.old_x = None
17     self.old_y = None
18     self.brush_size = 3
19     self.brush_color = "black"
20
21     # Store coordinates of drawn lines
22     self.line_coordinates = []
23
24     self.setup_ui()
25
26 def setup_ui(self):
27     # Main frame
28     main_frame = ttk.Frame(self.root, padding="10")
29     main_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N,
30                      tk.S))
31
32     # Control frame
33     control_frame = ttk.Frame(main_frame)
34     control_frame.grid(row=0, column=0, columnspan=2,
35                         sticky=(tk.W, tk.E), pady=(0, 10))
36
37     # Action buttons - all three buttons are always visible
38     ttk.Button(control_frame, text="Clear",
39                command=self.clear_canvas).pack(side=tk.LEFT,
40                                                padx=(0, 10))
41     ttk.Button(control_frame, text="Done",
42                command=self.done_drawing).pack(side=tk.LEFT,
43                                                padx=(0, 10))
44     ttk.Button(control_frame, text="Quit",
45                command=self.quit_app).pack(side=tk.LEFT, padx=(0,
46                                              10))
47
48     # Canvas
49     self.canvas = tk.Canvas(main_frame, bg="white",
50                            width=self.canvas_width,
51                            height=self.canvas_height,
52                            relief=tk.SUNKEN, borderwidth=2)
53     self.canvas.grid(row=1, column=0, columnspan=2, pady=(0, 10))
54
55     # Bind mouse events
56     self.canvas.bind("<B1-Motion>", self.paint)
```



DoodleBot Report

```
53     self.canvas.bind("<ButtonPress-1>", self.start_paint)
54     self.canvas.bind("<ButtonRelease-1>", self.stop_paint)
55
56     # Status label
57     self.status_label = ttk.Label(main_frame, text="Draw on the
58         canvas...")
59     self.status_label.grid(row=2, column=0, columnspan=2,
60         sticky=tk.W)
61
62     def start_paint(self, event):
63         self.old_x = event.x
64         self.old_y = event.y
65
66     def paint(self, event):
67         if self.old_x and self.old_y:
68             # Draw on canvas
69             self.canvas.create_line(self.old_x, self.old_y, event.x,
70                 event.y,
71                     width=self.brush_size,
72                         fill=self.brush_color,
73                         capstyle=tk.ROUND, smooth=tk.TRUE)
74
75             # Store line coordinates
76             line_coords = [self.old_x, self.old_y, event.x, event.y]
77             self.line_coordinates.append(line_coords)
78
79             self.old_x = event.x
80             self.old_y = event.y
81
82     def stop_paint(self, event):
83         self.old_x = None
84         self.old_y = None
85
86     def clear_canvas(self):
87         self.canvas.delete("all")
88         self.line_coordinates = []
89         self.status_label.config(text="Canvas cleared - ready for new
90             drawing")
91
92     def done_drawing(self):
93         # Save coordinates to JSON file without removing the Done
94             button
95         if self.line_coordinates:
```



DoodleBot Report

```
90     # Save coordinates to JSON file
91     self.save_coordinates_json()
92 else:
93     messagebox.showwarning("No Drawing",
94                           "Please draw something before
95                           clicking Done!")
96
97 def save_coordinates_json(self):
98     try:
99         # Ask user where to save the JSON file
100        file_path = filedialog.asksaveasfilename(
101            defaultextension=".json",
102            filetypes=[("JSON files", ".json"), ("All files",
103                      "*")],
104            title="Save coordinates as JSON"
105        )
106
107        if file_path:
108            # Create JSON data structure
109            coordinates_data = {
110                "drawing_info": {
111                    "canvas_width": self.canvas_width,
112                    "canvas_height": self.canvas_height,
113                    "total_line_segments":
114                        len(self.line_coordinates)
115                },
116                "line_coordinates": []
117            }
118
119            # Add all line coordinates
120            for i, coords in enumerate(self.line_coordinates):
121                line_data = {
122                    "line_id": i + 1,
123                    "start_point": {"x": coords[0], "y":
124                        coords[1]},
125                    "end_point": {"x": coords[2], "y": coords[3]}
126                }
127                coordinates_data["line_coordinates"].append(line_data)
128
129            # Save to JSON file
130            with open(file_path, 'w') as json_file:
131                json.dump(coordinates_data, json_file, indent=2)
```



```
129         success_msg = (f"Coordinates saved to JSON!\n"
130                         f"File: {file_path}\n"
131                         f"Total segments:\n"
132                         f"{len(self.line_coordinates)})")
133             messagebox.showinfo("Success", success_msg)
134
135             status_text = ("Drawing saved! You can continue\n"
136                            "drawing"\n
137                            "or save again.")
138             self.status_label.config(text=status_text)
139
140     except Exception as e:
141         messagebox.showerror("Error",
142                             f"Failed to save coordinates: {str(e)}")
143
144     def quit_app(self):
145         self.root.destroy()
146
147     def get_coordinates(self):
148         """Return the stored line coordinates"""
149         return self.line_coordinates
150
151     def main():
152         root = tk.Tk()
153         app = DrawingApp(root)
154         root.mainloop()
155
156 if __name__ == "__main__":
157     main()
```



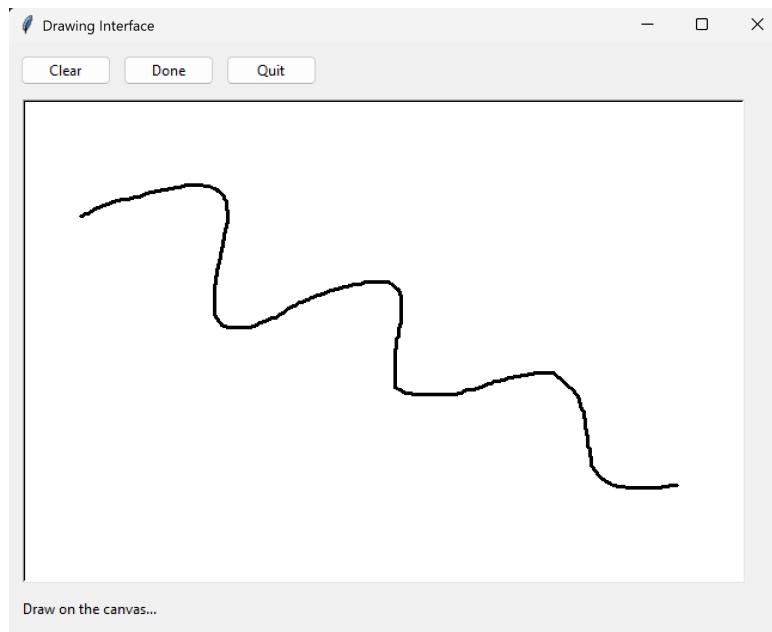


Figure 5: DoodleBot GUI

- **Logic-**

- Initialize GUI: It creates a window with a white canvas (600x400 pixels) and three buttons (Clear, Done, Quit)
- Mouse Click Detection: When user clicks on canvas, it records the starting position (x, y) coordinates
- Drawing Lines: As user drags mouse, it draws continuous lines by connecting previous position to current position
- Store Coordinates: Each line segment gets stored as [start_x, start_y, end_x, end_y] in a list called line_coordinates
- Real-time Display: Lines appear on screen immediately as user draws, with black color
- Clear Button: "Clear" button erases everything from canvas and empties the coordinate storage
- Done Button: "Done" button allows us to save the coordinates when user finishes drawing
- JSON Conversion: Drawing gets converted into structured JSON format with all line coordinates and canvas information

- File Save: I can choose where to save the JSON file containing all the drawing coordinate data

- **Configured a touch controller that interfaces with RPi**

- The GUI designed to accept the user-drawn sketch served as the primary user interface for the DoodleBot, allowing users to draw sketches that will be replicated by the robot.
- The touch-enabled LCD display interfaces directly with the RPi to capture user input and convert it into coordinate data for robot motion planning. We were able to run GUI on RPi and hence capture stroke data.

- **Assembly for 3-D printed parts**

- We got all the 3D printed components of the chassis of the DoodleBot and initiated assembling of the various components.

9 After Week-4

- **UDP Communication**

Logic- This function sends movement commands to a Raspberry Pi over UDP network communication:

- Purpose: Controls a robot's movement (forward, left, right, etc.) and pen state remotely
- Input: Takes command type (direction), speed value, and pen state (up/down)
- Data Format: Packages the parameters into a JSON dictionary with integer conversions
- Network Send: Transmits the JSON as UTF-8 encoded bytes via UDP socket to the RPi's IP/port
- State Tracking: Updates global current_pen_state variable to track pen position
- Error Handling: Catches and prints any UDP transmission failures for debugging

- **Problems Faced**

- PID control was causing the bot to move erratically.
- Some components such as motors with encoders were not available, and also the RPi camera got damaged.



- Issues in connecting the LCD screen to the RPi due to large number of jumper wires already connected to RPi.

- **Changes in the original design**

- Instead of PID control, corner-based steering was implemented
- The design was changed such that instead of using two RPis, one RPi and one Laptop have been used.
- As the RPi camera was not working, we used the phone-link option of Windows to use the camera of a phone.

10 Outcome

- Capturing user-drawn sketches via a touchscreen interface.
 - Converting those sketches into coordinate-based paths.
 - Mapping the paths to motion commands using image processing and ArUco marker tracking
 - Controlling a wheeled robot equipped with DC motor+wheel encoder with a pen-lift mechanism to draw the sketch on paper accurately
 - Executing smooth movements with PID-based motor control+encoder with RPi for precision.
- **Final code:** The code is in two parts- one part to be run in the laptop and another in the RPi.
Laptop Code:

Code working:

Lines 1-7:

- These lines import the necessary libraries. numpy is for numerical operations, cv2 is the OpenCV library for computer vision tasks, and cv2.aruco is a module within OpenCV specifically for ArUco marker detection.
- Socket, json, time, and threading are for network communication, data serialization, time-based operations, and running multiple tasks concurrently (though threading isn't used in the main loop, it's often a consideration for more complex projects).

Lines 11-12:



- These define the file paths for the camera's calibration data. Camera calibration is a crucial step to correct for lens distortion and accurately measure objects in the real world.

Lines 15-20:

- This section configures the ArUco marker system. It specifies the type of ArUco dictionary being used and sets up the detection parameters. It also defines the unique IDs for the four corner markers that form the drawing plane and the single marker on the robot itself.

Lines 23-31:

- This block defines the real-world dimensions of the drawing plane in centimeters. These values are used as a reference to map pixel coordinates from the camera's perspective into an accurate real-world coordinate system.

Lines 34-52:

- These are the key parameters for the robot's navigation. They include the camera index, the IP address and port for the UDP communication with the Raspberry Pi, and the various speeds and thresholds for the steering algorithm. The thresholds for turning and waypoint detection are particularly important for fine-tuning the robot's movement.

Lines 55-74:

- These are global variables used for managing the program's state. They track the camera calibration matrices, the robot's position, the user's drawn path (in both pixels and real-world centimeters), and the current state of the drawing process and path tracing.

Lines 77-83:

- This dictionary defines the visual and functional properties of the UI buttons displayed on the OpenCV window. This allows for interactive control without relying solely on keyboard shortcuts.

Lines 87-89:

- This sets up a UDP socket, which is a lightweight, connectionless protocol used to send commands to the Raspberry Pi. The IP address and port are specified to ensure the commands are sent to the correct device.

Lines 91-99 (load camera calibration pc):

- This function is responsible for loading the camera calibration data (camera matrix pc and dist coeffs pc) from the specified .npy files. This data is essential for correcting image distortion and performing accurate spatial calculations.



Lines 101-120 (send command to rpi):

- This function packages a command (e.g., "forward", "left", "stop"), speed, and pen state into a JSON message. It then sends this message via UDP to the Raspberry Pi. This is the core communication link between the PC's vision system and the robot's motors.

Lines 122-124 (calculate distance):

- A simple helper function to compute the Euclidean distance between two points in the 2D plane. This is used for checking if the robot has reached its target waypoint.

Lines 126-155 (is robot within boundaries):

- This function acts as a safety measure. It checks if the robot's current position is within the defined boundaries of the drawing plane (with a small margin). If the robot goes outside these bounds, an emergency stop command is issued to prevent it from driving off the table.

Lines 157-173 (calculate robot heading angle):

- This function calculates the robot's orientation in degrees. It uses the positions of the two front corners of the robot's ArUco marker to determine the heading vector and converts it to a normalized angle (0-360 degrees).

Lines 176-188 (calculate angle to goal):

- This function computes the angle from the robot's current position to the next waypoint. This information is crucial for the angle-based steering logic.

Lines 190-203 (angle difference):

- A utility function that finds the shortest angular distance between two angles. It returns a positive value for a right turn and a negative value for a left turn, which directly informs the steering logic.

Lines 205-302 (moveTowardsGoal):

- This is the heart of the navigation algorithm. It combines distance-based steering with angle-based steering for a more robust approach.
- It first performs a boundary check.
- It then checks if the robot has reached or overshot the current waypoint. The overshoot check is a new feature to prevent the robot from circling a target.
- It uses a multi-tiered approach:



- * If the angular error is large, it prioritizes a "fast turn."
- * If the angular error is moderate, it performs a "normal turn."
- * If the robot is roughly aligned, it switches to a distance-based approach, comparing the distances of its front-left and front-right corners to the goal.
- * If both angular and distance errors are small, it commands the robot to move "forward."

Lines 304-329 (get robot corner positions cm):

- This function takes the pixel coordinates of the robot's ArUco marker and uses the inverse homography matrix to transform them into real-world centimeter coordinates. It returns the positions of the robot's center, left-front, and right-front corners.

Lines 331-403 (mouse callback):

- This function handles all mouse interactions with the control window. It checks if the user clicked on a UI button (Quit, Save, Clear) or if they are drawing a path. When a path is drawn, it stores the discrete pixel points. Clicking "Save" or pressing "Enter" triggers the conversion of these pixel points into a list of real-world waypoints in centimeters.

Lines 406-430:

- This is the program's entry point. It loads the camera calibration, initializes the camera capture, and creates the OpenCV window. It also sets up the mouse callback function to handle user input.

Lines 439-444:

- The `while not app_should_quit:` loop is the main execution block of the program. It continuously reads frames from the camera. If a frame fails to grab, it waits a moment before trying again.

Lines 446-447:

- The grabbed frame is copied to a display frame and then undistorted using the loaded camera calibration matrices. This is a vital step for accurate measurements.

Lines 449-478:

- This block is responsible for Homography Calculation.
- It detects all ArUco markers in the frame.
- It then checks if all four plane corner markers (PLANE_CORNER_IDS) are detected.



- If they are, it calculates a homography matrix (H). This matrix is the key to converting pixel coordinates on the camera feed into real-world centimeter coordinates. It essentially "flattens" the camera's perspective view.
- It also calculates the inverse homography matrix, which is used to perform the reverse conversion (pixels to CM). If not all markers are detected, it uses the last known homography to continue operations.

Lines 481-502:

- This visual section draws a grid on the display frame to represent the real-world coordinate system in centimeters. This helps the user visualize the drawing plane and confirm that the homography calculation is working correctly.

Lines 505-543:

- This block handles the detection and visualization of the robot marker (BOT_MARKER_ID).
- If the robot's marker is found and the homography is valid, it calls `get_robot_corner_positions_cm` to get the robot's real-world position and orientation.
- It then draws visual elements on the display frame to represent the robot's position, front corners, and heading, providing real-time feedback to the user.

Lines 546-571:

- This part of the code visualizes the user's input. It draws the path the user has clicked on (drawing pixel points) and, if tracing is active, draws the path that has already been completed and highlights the current target waypoint.

Lines 574-610:

- This is the core Steering Control Logic.
- If path tracing is active and the robot's position is known, it gets the next waypoint.
- It then calls the `moveTowardsGoal` function at a controlled interval (COMMAND_INTERVAL).
- Based on the return value of `moveTowardsGoal`, it either advances to the next waypoint, stops due to an out-of-bounds error, or completes the trace.

Lines 613-670:

- This section manages the user interface and displays status information directly on the video feed. It shows the current state (e.g., "Idle," "Tracing"), the robot's coordinates and heading, and the number of drawn points and waypoints. It also draws the UI buttons defined earlier.



Lines 674-706:

- This handles keyboard inputs. Pressing Enter converts the currently drawn pixel points into a set of waypoints in real-world centimeters and starts the tracing process. Pressing 's' starts tracing a path that has already been converted and saved.

Lines 708-715:

- The `finally` block ensures that regardless of how the program exits (either by user command or an error), it performs a graceful shutdown. It sends a final "stop" command to the robot, releases the camera, closes the OpenCV windows, and closes the UDP socket. This prevents the robot from continuing to move unintentionally.

RPi Code:

Explanation:

Lines 1-20:

- Imports necessary libraries like `json`, `socket`, and `threading`. It then defines the GPIO pin numbers for the two motors (Left and Right), which are controlled by an L298N motor driver. The pins for motor direction (`IN1`, `IN2`, `IN3`, `IN4`) and speed control (`ENA`, `ENB`) are specified.

Lines 23-34:

- Sets up the Raspberry Pi's GPIO pins according to the pin numbers defined above.
- Initializes the PWM (Pulse Width Modulation) for the speed control pins (`ENA` and `ENB`) with a frequency of 100 Hz. This allows the motor speeds to be controlled precisely.

Lines 37-41 (set motor speed):

- This function sets the speed of a motor by changing the duty cycle of its PWM pin. The speed value is clamped between 0 and 100 to prevent errors and ensure safe operation.

Lines 43-50 (stop motors):

- A crucial safety function that sets all motor direction pins to LOW and all PWM speed pins to a 0% duty cycle, ensuring the motors immediately stop.

Lines 52-124 (execute movement command):

- This is the main motor control function. It takes a command string (e.g., "forward", "left") and a speed value.
 - * For "forward", it sets both motors to spin forward at the specified speed.



- * For "left" and "right" commands, it implements gradual turns by slowing down one motor while the other runs at a higher speed.
- * For "fast left" and "fast right" commands, it executes sharp turns by spinning one motor forward and the other backward, causing the robot to pivot in place.

Lines 126-136 (control pen):

- This function is a placeholder for controlling the pen mechanism. It is designed to take a state (0 for up, 1 for down) and would be expanded to control a servo or solenoid to lift and lower the pen.

Lines 139-166 (receive commands udp thread):

- This function runs in a separate thread. It creates a UDP socket and listens for incoming data from the PC on the specified port. When a message is received, it decodes the JSON data and adds it to a command queue to be processed by the main loop. It uses a threading.Lock to ensure that the queue is accessed safely from both the main thread and the receiving thread.

Lines 169-171 (main):

- The entry point of the script. It starts the receive_commands_udp_thread in the background.

Lines 178-212 (Main Loop):

- The main thread enters a loop where it checks the command queue for new messages. It safely retrieves all waiting commands, clears the queue, and then iterates through them.
- For each command, it attempts to parse the JSON data and calls execute_movement_command and control_pen with the received parameters.

Lines 214-224 (Cleanup):

- The try...finally block ensures that if the program is interrupted (e.g., by Ctrl+C or an unexpected error), it executes a cleanup routine. This routine calls stop_motors() to prevent the robot from running unintentionally, stops the PWM signals, and releases all GPIO pins with GPIO.cleanup(). This is critical for preventing resource conflicts and ensuring the hardware is in a safe state.



11 Mentors

- Ashish Upadhyay (230231)
- Bhuvan Kumar K P (230297)
- Dhakshith Sureshkumar (230359)
- Mrigeesh Ashwin K (230666)

12 Mentees

- Akash Kumar (240078)
- Akriti yadav (240082)
- Ananthan R (240119)
- Ayush Muttepawar (240243)
- Jugal Pahuja (240501)
- Manish Kumar Meena (240623)
- Pratham Gupta (240779)
- Sarthak Sehgal (240941)
- Shubham (241007)
- Sivaganesh B(241019)

