DEPARTMENT OF PHYSICS
UNIVERSITY OF COLOMBO

PH 3032 - Embedded Systems Laboratory

Hand Tracking Robot Arm

Name: Saumya Induwara Jayasinghe

Index: 16095

Registration number: 2021s18825

Table of Contents

Abstract

The development of a hand-tracking robotic arm using an ATmega328P microcontroller represents an innovative approach to human-computer interaction by mimicking human hand movements in real-time. This project integrates multiple technologies, including Python-based hand tracking, Bluetooth communication, and embedded C programming for microcontroller control. Using a webcam, a Python script detects key hand landmarks in three-dimensional space, extracts their coordinates, and transmits them via Bluetooth to the microcontroller. The ATmega328P processes this incoming data to control five servo (can be extends to customized number of servos) motors on a robotic arm, allowing it to replicate the movements of a human hand.

One of the main challenges in designing such systems lies in achieving real-time data transmission and precise servo control, both critical for creating a seamless mimicry. The system's accuracy, responsiveness, and minimal latency demonstrate the potential of embedded systems in robotics and remote-control applications. By effectively combining Bluetooth data transfer with real-time motor control, this project can serve as a foundation for numerous applications, such as assistive technology for individuals with disabilities, remote-controlled robots for hazardous environments, and virtual reality simulations where users interact with physical objects through gestures.

This project exemplifies how embedded systems, combined with vision-based tracking and wireless communication, can provide intuitive interfaces for robotics. The ability of the robot arm to mirror human hand movements opens possibilities in fields like telemedicine, where remote manipulation of objects is essential, and industrial automation, where intricate, remote control of machinery is often required. This project is a steppingstone toward more advanced gesture-controlled systems and serves as an example of how low-cost microcontrollers like the ATmega328P can be leveraged to create impactful solutions.

1.Introduction

In recent years, robotics and embedded systems have been pivotal in shaping various technological advancements, especially in the areas of automation and human-machine interaction. One such advancement is the creation of systems that allow for gesture-controlled robotics, where humans can remotely control devices and manipulate objects using hand movements. This project explores the development of a robotic arm controlled by real-time hand tracking using web cam live, utilizing the ATmega328P microcontroller, Python-based computer vision, and Bluetooth communication to provide a responsive, gesture-based interface.

Hand-tracking is implemented through Python library called MediaPipe, which allow landmark detection of key hand points (20 Hand landmarks included) and extraction of their coordinates. These coordinates are then filtered out and transmitted to the ATmega328P microcontroller via a Bluetooth module (HC-05 or HC-06), making it possible for the robotic arm to perform precise, real-time movements. Bluetooth provides a convenient, wireless interface, allowing the user to control the robotic arm from a short distance while avoiding the limitations of wired systems. In this project, five servos are configured to replicate wrist, finger, and elbow motions, allowing the robotic arm to follow the movements detected by a webcam on the user's hand.

This project highlights the potential of combining embedded systems with computer vision and wireless communication. The applications of a hand-tracking robotic arm extend into several fields. In industrial settings, it could enable operators to control robotic tools remotely, reducing the need for direct interaction in hazardous environments, thereby enhancing worker safety. In the medical field, such systems could assist surgeons in performing delicate, remote operations with precision, providing better accessibility in areas lacking specialized medical expertise.

In military applications, a hand-tracking robotic arm could allow personnel to handle and manipulate dangerous or sensitive equipment from a safe distance, such as controlling bomb disposal robots or interacting with explosive devices, thus reducing risk to human life. The intuitive hand-based control could also aid in operating remote-controlled drones or reconnaissance equipment in high-risk zones, where precise and responsive maneuvering is crucial.

For daily use, gesture-controlled robotic arms could be integrated into smart homes, enabling users to perform tasks like adjusting appliances, picking up objects, or aiding in kitchen work hands-free. In assistive technology, individuals with disabilities could employ these gesture-based robotic arms as auxiliary devices to perform routine tasks, offering increased independence and control. Additionally, in fields like virtual reality and gaming, the robot arm could be used to provide haptic feedback, creating a more immersive experience by allowing users to physically interact with virtual environments. Overall, this project serves as both a technical accomplishment in embedded systems and robotics and a proof of concept for applications across multiple industries, from safety and healthcare to everyday convenience and beyond.

2.Methodology

2.1 Hardware Setup

Complete components used for the Robot arm as follows.

Components List

1. ATmega328p Micro-controller
2. SG90 Servos(3x)
3. MG996R Servos(2x)
4. HC-05 Bluetooth Module
5. LM2596 DC-DC step-down buck converter (To power the microcontroller)
6. XL4015 DC-DC step-down buck converter (To power the servos)
7. 3.7V 18650 Li-ion rechargeable batteries(3x)

2.1.1 Custom designed robot arm rig

The robot arm was designed using the open-source 3D modeling software, Blender. The 3D model consists of ten custom-designed parts, specifically made to fit the servo types listed in the components. The model was intentionally kept as simple as possible to allow for future modifications; for instance, the gripper was designed based on a basic robotic hand gripper concept. Finally, the parts were 3D printed using white PLA filament.

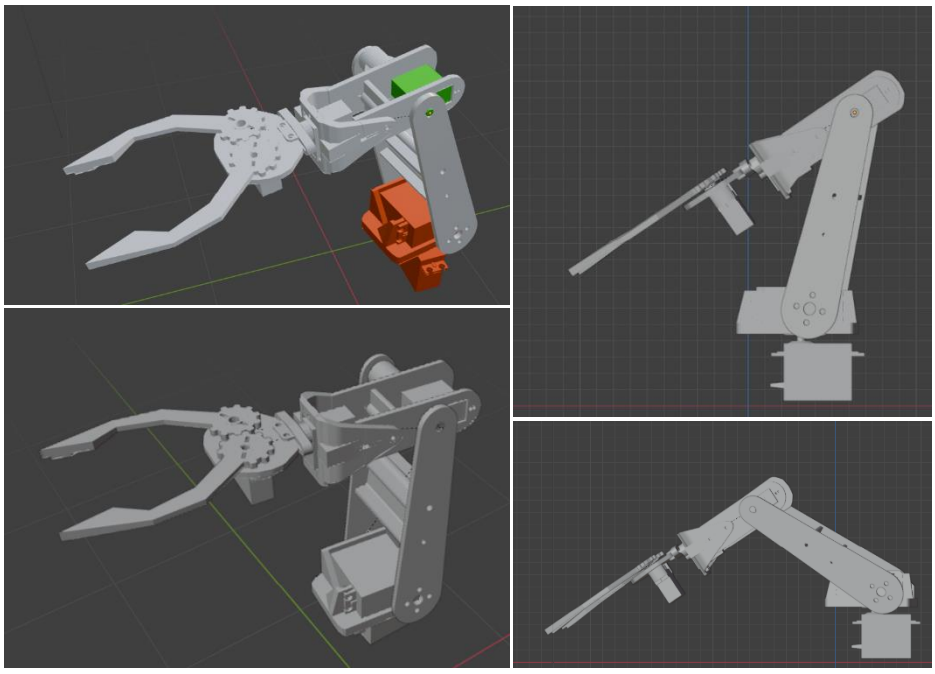

Figure 1: 3D model of the Robot arm rig (In Blender 3D workspace).

2.2 Hand Tracking and Coordinate Acquisition

2.2.1 Hand Tracking

Hand tracking is accomplished using the MediaPipe Python library, an open-source framework developed by Google that enables efficient, real-time hand tracking and highly accurate landmark detection. In Python, MediaPipe offers pre-trained models capable of detecting 21 key hand landmarks, including fingertips and major joints, from a live video feed or still image. The hand-tracking module operates by first identifying the hand's position and orientation, then extracting precise coordinates for each landmark point. The framework employs machine learning algorithms to accurately recognize these landmarks, even under varying lighting conditions or different hand positions, making it particularly suitable for real-time applications such as gesture control, augmented reality (AR), virtual reality (VR), and robotics. By providing detailed and consistent hand data, MediaPipe significantly simplifies complex gesture recognition tasks in Python, enabling seamless interaction between humans and machines.

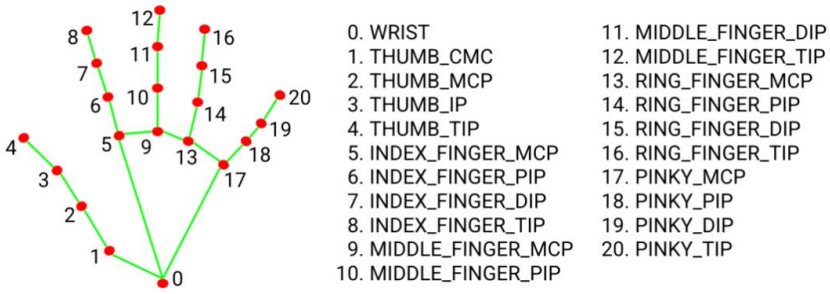

Figure 2: Hand Landmarks of the MediaPipe Hand Landmarker model(full).

Using the given detectable hand landmarks, robot arm uses only 9 landmarks IDs for its operations (can be extended for all 21 landmarks according to the requirements). From the 9 IDs, 7 were used to directly control the robot arm movements while other 2 IDs were used for the stabilization process of the entire hand my neglecting unnecessary movements (filtration techniques are discussed later in this section)

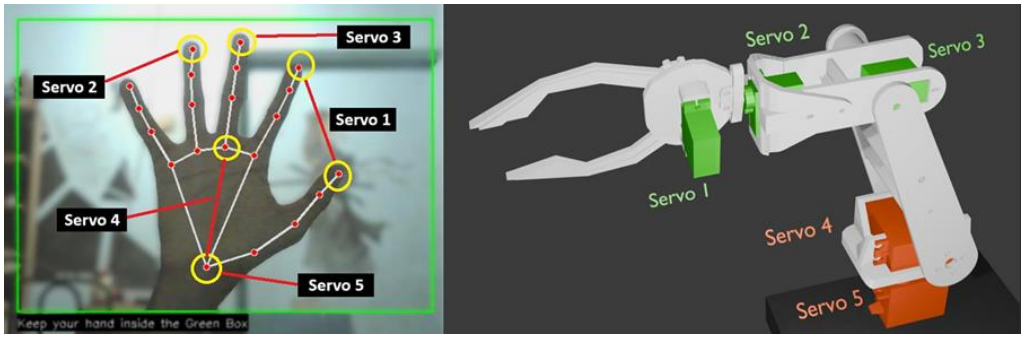

Figure 3: Hand Landmarks IDs used(left) and corresponding servo in the arm rig(right).

2.2.2 Hand Landmark Coordinate Extraction.

Basic structural coordinates are tracked and extracted by calling the MediaPipe library within the Python script. To manage the live feed from the webcam, the OpenCV (CV2) library is utilized. The current system is designed to track only one hand at a time to minimize unnecessary movements and reduce jittering. To optimize this objective, several key parameters related to the tracking process are defined as follows,

```
mp_hands = mp.solutions.hands

hands = mp_hands.Hands(static_image_mode=False,

                       max_num_hands=1,

                       min_detection_confidence=0.5,

                       min_tracking_confidence=0.5)

mp_drawing = mp.solutions.drawing_utils
```

Code snippets provide as above shows the initialization of Media Pipe hand landmarker model. Since we are gaining live feed from the web cam *static_image_mode* is set to *false.* Other parameters values are assigned for better tracking performance for the web cam live feed.

After tracing the hand, only 9 coordinates are extracted. Each hand landmark coordinate is composed of x and y values (not including the depth or z) relative to the window size of the web cam (here, window size is 640px width and 480px height).

```
if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(image, hand_landmarks, mp_hands.HAND_CONNECTIONS)

            lm_thumb = hand_landmarks.landmark[4]
            lm_index = hand_landmarks.landmark[8]

            lm_middle = hand_landmarks.landmark[12]
            lm_middle_base = hand_landmarks.landmark[9]

            lm_ring = hand_landmarks.landmark[16]
            lm_ring_base = hand_landmarks.landmark[13]

            lm_small = hand_landmarks.landmark[20]
            lm_small_base = hand_landmarks.landmark[17]

            palm_base = hand_landmarks.landmark[0]
```

In Mediapipe, after calling landmark IDs we can simply extract the relevant x,y coordinates for each IDs separately. As an example, below code part shows how we can get x,y values by knowing our given but fixed window size of our web cam window while tracking the hand.

```
palm_base = hand_landmarks.landmark[0]

cy_base= int(palm_base.y*h) # h is the window height
cx_base= int(palm_base.x*w) # w is the window width
```

2.2.3 Coordinates Filtration

Mediapipe handlandmarker model is excellent at tracking predefine hand landmarks in a great way of accuracy in several lighting conditions. But when the issue arises when those coordinates are mapped to the servos positions since they have different minimal tolerance levels. In simple terms, small movement of the hand is properly tracked by the handllandmarker model with some jittering, so this jittering is enlarged when it mapped to tilting angles for each servo. So, the original coordinates which are mapped is subject to a filtration process such unnecessary movements which are existing in between define tolerance level is ignored and only the actual movement of the hand is given as the coordinate values. Those coordinates values are then converted as the degree value for each servo to tilt.

During the filtration, objective is to respond for the sudden movements and slow movements of the hand and receive the coordinates and ignore the high frequency (or noisy) movements caused by other external factors (web cam faults, low lightings) when there are no real hand movements. So, one of the best filtering techniques is to use EMA (Exponential Moving Average) filter. EMA filter allows us to generate a moving average based on most recent data and history of the data for a given set of data chunk and it responsible to react the next data very quickly or slowly based on our threshold value and alpha value(smoothing factor).If we set our smoothing factor(between 0-1 typically) as 0.1 , then the filter responsive to the history larger than the current data, so we can have a better average value.

When we consider the threshold value for each filter, values were based on custom settings done experimentally using the hand tracking model + servo angles, so the threshold values are differed from each other for each landmark ID for the hand tracking model.

The following graph shows the x coordinates variation of the thumb fingertip (right or left hand) in the hand tracking window for 200 seconds of time, during this time the thumb tip moved rapidly and smoothly to get the x coordinates and then filtered using ordinary moving average filter and median filter techniques.

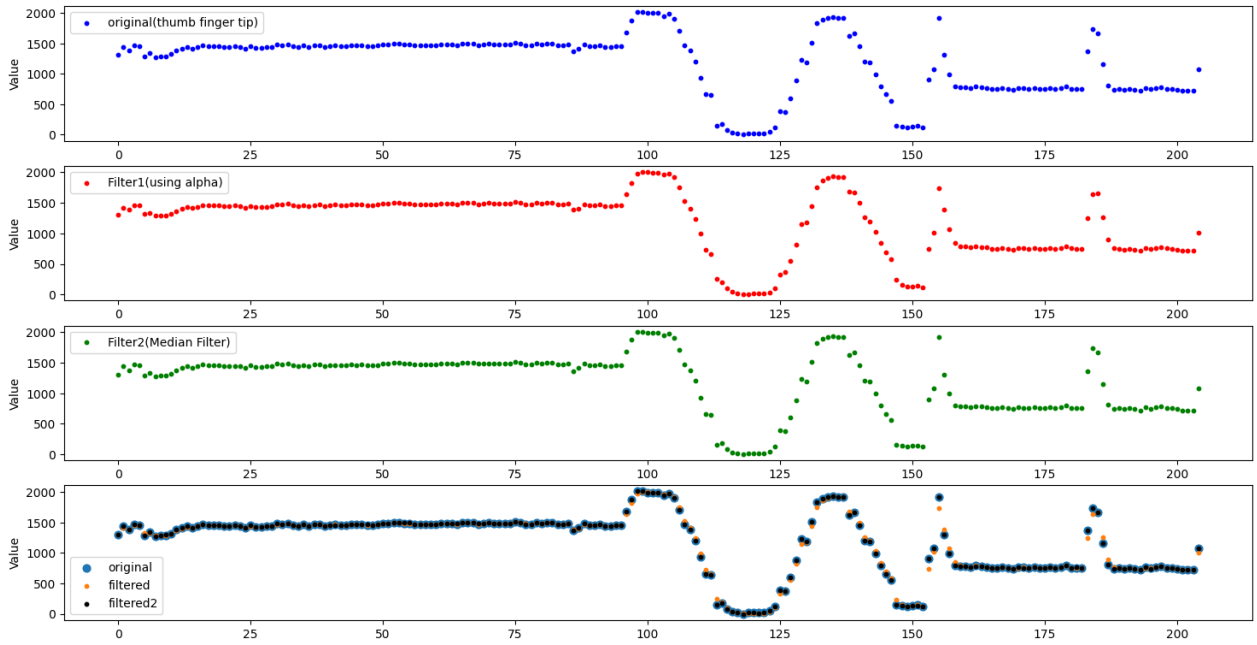

Figure 4: Different filter techniques applied for certain landmark coordinates.

So, if we change the alpha (smoothing factor) for the ordinary moving average filter or median filter, smoothing objective may complete but they unable to give a quick response to the current data change, so the instantaneous movement of the hand is not traceable even if the hand landmarker model gives the tracked coordinates. Then as previously mentioned, EMA filter is applied.

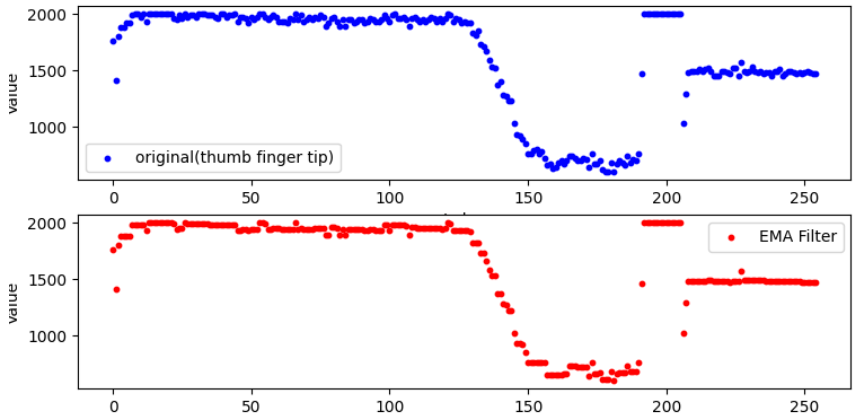

Figure 5: EMA filter applied for a hand landmark for 250 seconds.

As the next step, EMA filter(with customized alpha value) checked to its response for the sudden movements and variations of the x(or y) coordinate for a certain hand landmark.

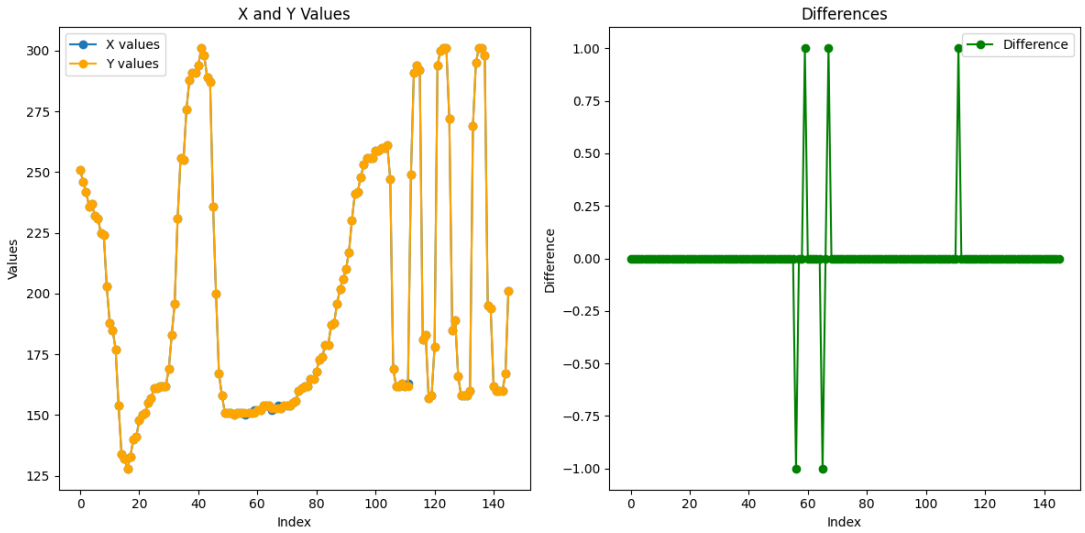

Figure 6: Response check for the EMA filter

In above graph, **x values** related to the original x coordinate of the thumb fingertip and **y values** represents the filtered x coordinates taken for 150 seconds. So the left graph (difference graph) shows how much the change happen after filtration done using EMA filter along with the quick and smooth variations of the original coordinate values, but the most of the time difference is negligible and EMA filter has a great response to quick change and small changes happen to original data.

Python code for customized EMA filter,

```
previous_ema = None
def smoothyy(data_point, previous_ema=None, smoothing_factor=0.1):

    if previous_ema is None:
        # Initialize EMA to the first data point if no previous EMA exists
        previous_ema = data_point
    else:
        # Update the EMA based on the new data point
        new_avg = (smoothing_factor * data_point) + (1 - smoothing_factor) * previous_ema

        if data_point > previous_ema + 50: # 50 is the custom threshold for thumb finger
            new_avg = data_point
        elif data_point < previous_ema - 50:
            new_avg = data_point

        previous_ema = new_avg

    return int(previous_ema)
```

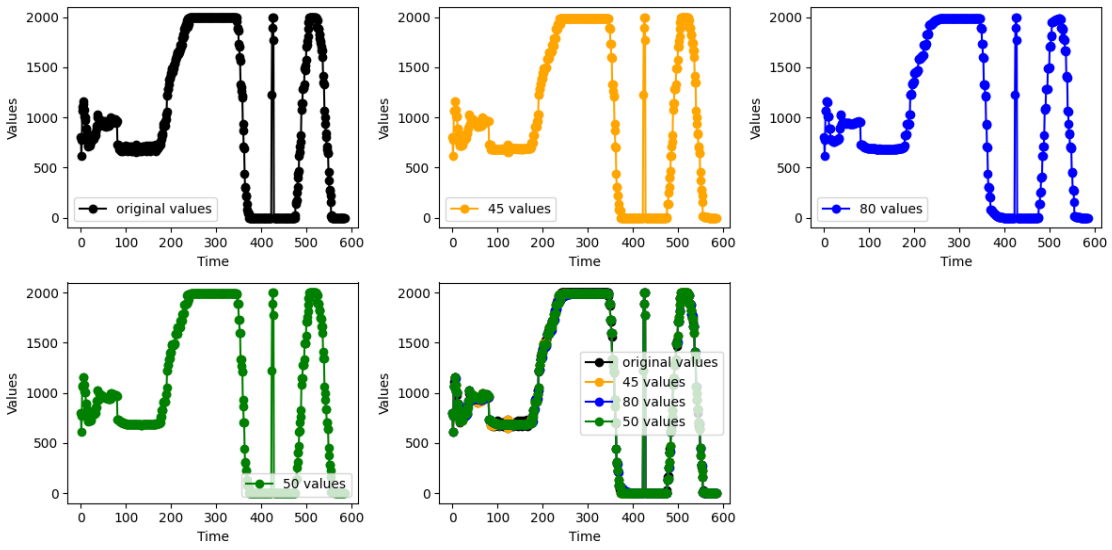

Figure 7: EMA filter used for 4 different hand landmarks with relevant threshold values.

2.2.4 Coordinate Processing and Mapping

9 Hand landmark coordinates are taken by their hand landmark IDs. Each x,y pair for the hand landmark ID is then used for the mapping process to servo angles. Before mapping, those x and y coordinates are used to create some custom effects for the hand tracking process.

Here, our robot hand does not want to react when the hand is rotated, so to do that, hand landmark IDs 0,7,9 and 13 are used are reference point for hand landmark IDs 4,8,12,16,20.

Landmark IDs 0,7,9,13 is not changing since they are the bases for palm, middle finger, ring finger and small finger. ID 0 is used along with ID 9 which is landmark for base of the middle finger to make a constant length parameter which is used to neglect the hand rotation and forward and backward movements (for entire hand).

```
#scale constant
            #using base and middle finger base
            x0 = (cx_middle_base-cx_base)** 2
            y0 = (cy_middle_base-cy_base)** 2
            main_diff = int(np.sqrt(x0 + y0))
            centered_diff = abs(main_diff-150) #150 is the centered scale factor
```

The following python code snippets show the x,y coordinate processing for thumb fingertip, middle fingertip, ring fingertip and small fingertip which are stabilized using above technique.

```
            x1 = (cx_index-cx_thumb)**2
            y1 = (cy_index-cy_thumb)**2
            grip_diff = int((np.sqrt(x1 + y1)) + centered_diff)


            #wrist rotating.........by small finger
            x2 = (cx_small-cx_base)**2
            y2 = (cy_small-cy_base)**2
            small_diff = int(np.sqrt(x2 + y2)) + centered_diff

            #elbow rotating........by ring finger
            x3= (cx_ring-cx_ring_base)**2
            y3 = (cy_ring-cy_ring_base)**2
            ring_diff = int(np.sqrt(x3 + y3)) + centered_diff

            #shoulder rotating........by middle finger
            x4= (cx_middle-cx_middle_base)**2
            y4 = (cy_middle-cy_middle_base)**2
            middle_diff = int(np.sqrt(x4 + y4)) + centered_diff
```

After the coordinates are processed, they mapped to some range due to need of angle generation for servos to tilt. Tilting angles of the servos are based on the PWM pulse width generated. So, could be easier if we can supply the coordinates directly as pulse width values. But the PWM pulse defined for servos are differ from servo manufacturer types. Most of the servos tilt for 90° for 1ms of pulse (for continuous servos the situation is bit different).

```
def map_range(value, from_min, from_max, to_min, to_max):
    """ Map a value from one range to another. """
    if value<50:
        value=50
    if value>200:
        value= 200

    return int((value - from_min) * (to_max - to_min) / (from_max - from_min) + to_min)

def map_range2(value, from_min, from_max, to_min, to_max):
    """ Map a value from one range to another. """
    if value>500:
        value= 500
    if value<200:
        value= 200

    return int((value - from_min) * (to_max - to_min) /
               (from_max - from_min) + to_min)
```

Mapping functions define by setting cutoff levels to ignore undefine hand locations beyond the hand tracking windows. Mapping function *map_range* is used for thumb finger, index finger, middle finger, ring finger and small finger. Mapping function *map_range2* is used for wrist coordinates.

2.2.5 Coordinate Transmission

Robot arm is working with 5 servos, then we must update 5 separate variables. And the objective is to operate every servo at once when the relevant variable for the servo angle is updating. Here, for the function of the robot arm, following basic setup is used,

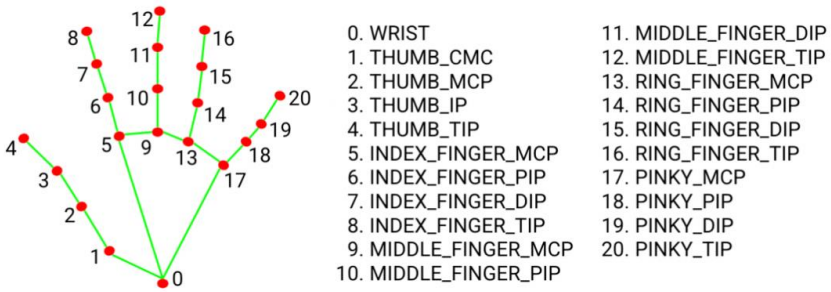

Figure 8: Hand Landmark IDs

1. Coordinate difference between INDEX_FINGER_TIP and THUMB_FINGER_TIP tip is used to open and close the gripper of the robot arm.
2. Coordinate difference of the MIDDLE_FINGER_TIP and WRIST is used for the elbow movements of the robot arm.
3. Coordinate difference of PINKY_TIP and WRIST is used for shoulder movements for the robot arm.
4. x coordinate of the wrist is used for horizontal movements of the entire robot arm
5. y coordinate of the wrist is used for wrist movement of the robot arm.

Therefore totally 5 separate coordinates values have to transmit, and each coordinate is updating instantly with hand tracker model.

```
data = f"{thumb_and_index},{middle},{pinky},{wristX},{wristY}\n"
```

2.3 Data Processing on ATmega328p

Robot arm should be good enough to respond sudden and fine movements of the human hand, then the microcontroller should handle each task parallelly and quickly by reducing errors. To optimize the speed and the functionality of the ATmega328p micro-controller, it has fuse burned to external 16Mhz oscillator. This process helps to manage and generate clean software PWM pulses, proper communication using USART module and efficient data processing (since we have 5 separate coordinates being transmitted via serially).

2.3.1 Serial Communication with USART

Since we are using HC-05 Bluetooth module with the robot arm, for serial communication the internal USART module of the ATmega328p micro-controller can be used very efficiently. Here, basic setup used for USART module with baud rate 9600 and since our micro-controller is fused burned to external 16Mhz crystal oscillator, higher baud rates can be used. But the recommended and most stable baud rate for HC-05 module is 9600.

USART basic initialization setup code in C,

```
void USART_Init(unsigned int ubrr) {

    UBRR0H = (unsigned char)(ubrr >> 8);
    UBRR0L = (unsigned char)ubrr;
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
}
```

After initialization of the USART, two custom C functions used to transmit and receive character type data.

```
void USART_Transmit(unsigned char data) {

    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = data;
}

unsigned char USART_Receive(void) {
    while (!(UCSR0A & (1 << RXC0)));
    return UDR0;
}
```

The void type of function USART_Transmit is used to write the character type data to UDR0 register when the communication is enabled. And the USART_Recieve function used to read the character data when the communication is enabled and UDR0 is available.

2.3.2 Coordinate Separation for Generating PWM pulses.

ATmega328p receives a data set of 5 processed coordinates via Bluetooth using USART module and then next step is to sperate each coordinate for the angle conversion for each servo. By USART, default we receive character type data and now we want integer type(basically) to manipulate custom PWM pulses between 0 to 1ms (here 0 means 0 and 1ms means 2000 after mapping process).

Example data set: 1030,250,1400,432,834

```
  char buffer[500];  //buffer to hold the received string-->data bit size is about 250 bits enough
then

    uint8_t index = 0;
    uint8_t count = 0;  //To count the number of numbers processed
    int  numbers[5];
    //buffer flushing
    memset(buffer, 0, sizeof(buffer));

    while (1) {
        char received = USART_Receive();
        if (received == ',' || received == '\n') {
            buffer[index] = '\0';  //Null-terminate the string
            if (count < 5) {
                numbers[count++] = atoi(buffer); //Assigning and converting the string to an integer
            }
            index = 0;  //reset the buffer index for the next number
            //If a newline character is received, process the numbers
            if (received == '\n' && count == 5) {
                count = 0;  //reset count for the next set of 5 numbers
            }
        } else {
            buffer[index++] = received;  //store the character in the buffer
        }

        USART_Transmit(received);
    }
```

Using the above C code, characters stored as numbers are separated by comma and converted back to integer form for the angle calculation process for servo. In the above code, char type variable *buffer* is used to store whole data set (as in example data set) when it is transmitted and received by USART module, but after every last coordinated processed, the buffer is flushed to improve the speed of the data processing.

After separation by commas, each number is stored in int type array called *numbers.* Then those values can be used directly as PWM pulse width for relevant servo.

numbers[0] element related to servo 1 (SG90 servo)

numbers[1] element related to servo 2(SG90 servo)

numbers[2] element related to servo 3(SG90 servo)

numbers[3] element related to servo 4(MG996R servo)

numbers[4] element related to servo 5(MG996R servo)

2.3.3 Custom PWM pulses for 5 servos

ATmega328p micro-controller has 3 separate timers (timer0, timer1 and timer2). By default, each timer only provides two distinct PWM outputs and then total 6 PWM outputs. But the timer0 and timer2 are 8-bit implies that their PWM pulse frequency can be differ from the PWM pulse generating frequency of the timer1. On the other hand, for the better functioning of servos, they require 50Hz PWM pulses, so from above 3 default timers, only the timer1 can give 50Hz clean PWM pulse but the output is limited to 2.

Therefore, the goal is to create 5 distinct 50Hz PWM pulses. To do that, timer1 is used and the reason is based on timer1 special offers available on timer/counter1 module of ATmega328p micro-controller. Timer1 is switched to fast PWM mode by setting TOP value to ICR1 by assigning ICR1 value. Timer1 is presale to 8 such that to get optimized resolution for counting process.

```
void init_timer1_servo() {

    //Timer1 to Fast PWM modewith ICR1 as top value
    TCCR1A |= (1 << WGM11);
    TCCR1B |= (1 << WGM12) | (1 << WGM13) | (1 << CS11); //Prescaler 8

    ICR1= 7999; //Customized value here....
}
```

ICR1 value is set to 7999. This value is based on the fact that, we have to create 5 different PWM pulses which are 50Hz frequency and the clock speed of fuse burned ATmega328p is 16Mhz as mentioned.

ICR1 value = 7999 creates a total of 4ms time range for timer1 to count from 0 to 7999, after that timer1 overflow vectors and timer1 compare matchA vectors are used to identify the overflows and relevant compare matches happened to create next steps. Basic flow of custom PWM cycle for 5 separate PWM as below,

1. PWM output pins used here is from PORTB pins, PB0, PB1, PB2, PB3 and PB4(can be extended to other ports also)
2. At time t=0, PB0 is set to HIGH(Output) while all other pins set to LOW when timer1 starts to count and 1st value which is stored in an array related to compare match values are assigned to OCR1A register
3. During timer1 counting to 7999, when compare match happen the PB0 is set to LOW.
4. When timer1 reaches 7999 value, overflow vector arises and using interrupt routine PB1 is set to HIGH while all other pins set to LOW (PB0 is already LOW).
5. When PB1 set to HIGH, OCR1A register then remove its old value and then assign the 2nd value for compare matches.
6. Above steps continue up to PB4 and at that time 20ms spent and automatically created PWM pulses for each PORTB port provided with the frequency of 50Hz for each.

Timer1 overflow and compare matchA interrupt service routine usage.

```
ISR(TIMER1_COMPA_vect) {

    PIN_LOW(PORTB, currentChannel);  //setting the current pin low when OCR1A is reached
}

ISR(TIMER1_OVF_vect) {

    //updating the next channel
    currentChannel++;
    if (currentChannel >= NUM_CHANNELS) {
        currentChannel = 0;  //first channel
    }

    set_pulse_width(array1[currentChannel]);
    PIN_HIGH(PORTB, currentChannel);  //set the new pin high at the start of the cycle

}
```

3.Implementation

3.1 Power Requirements

Servos are power hungry for its operation, then power should be supplied continuously for better performance. Here, for the robot arm uses 3.7V li-ion 18650 type batteries of current capacity of 4200mAh. Usage of 3 batteries handles power problem very well for all servos as well as for the ATmega328p microcontroller. Two separate DC-DC buck converters are used to step down the voltages to power 5 servos and microcontroller separately.

ATmega328p power usage

Active mode (at 16Mhz, 5V): 15-20 mA

HC-05 Bluetooth module power usage (at 5V)

Active Mode (while transmitting): 30–40 mA.

Servo power usage

1. SG90 Micro servo
    - **Voltage Range**: 4.8V to 6V
    - **Idle Current**: Around 10 mA
    - **Operating Current**: 100–250 mA under normal load
    - **Stall Current**: Around 650 mA at 5V (when the servo is held in place and unable to move further)
2. MG996R servo (Standard high-torque servo)
    - **Voltage Range**: 4.8V to 7.2V
    - **Idle Current**: Around 10 mA
    - **Operating Current**: 500–900 mA under normal load
    - **Stall Current**: 2.5A at 6V

3.2 Robot Arm Structure

Robot arm rig is consisting of 5 distinct parts which are controlled by each servo. Each part has its own degree of freedom and constrained by the angle of rotation.

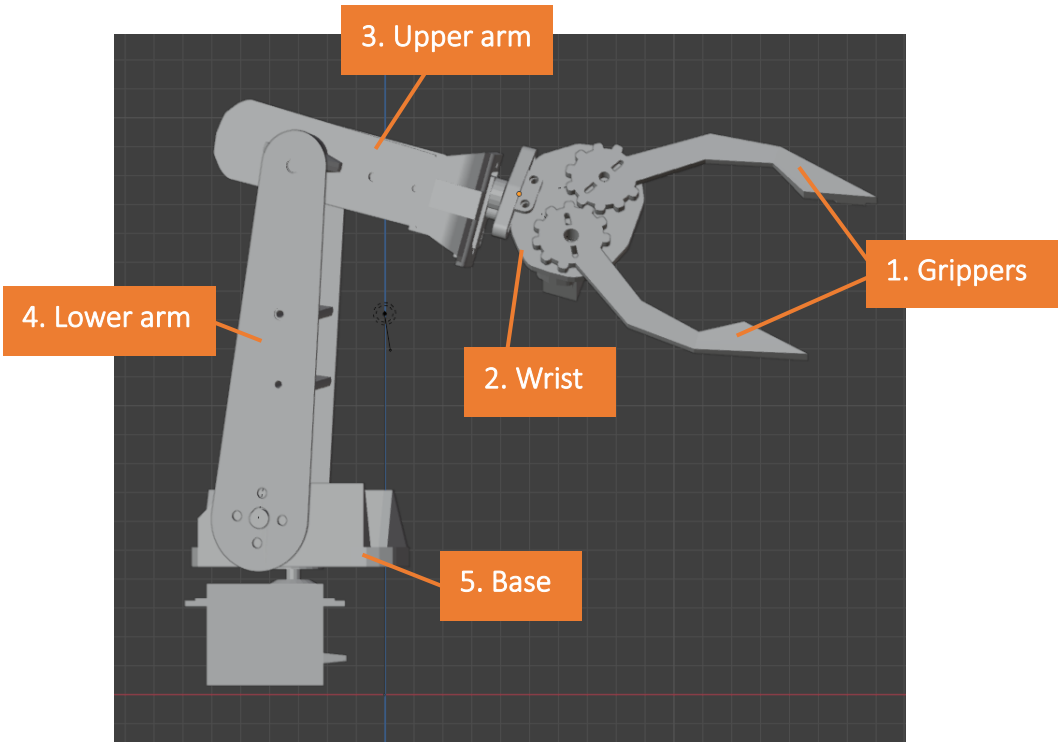

Figure 9: Structure of the robot arm

Rotating freedom of each segment of the rig is limited by the code(limits cause by the physical structure of the robot arm is not calculated).

Grippers : 90°

Wrist : 90°

Upper arm : 60°

Lower arm : 60°

Base: 180°

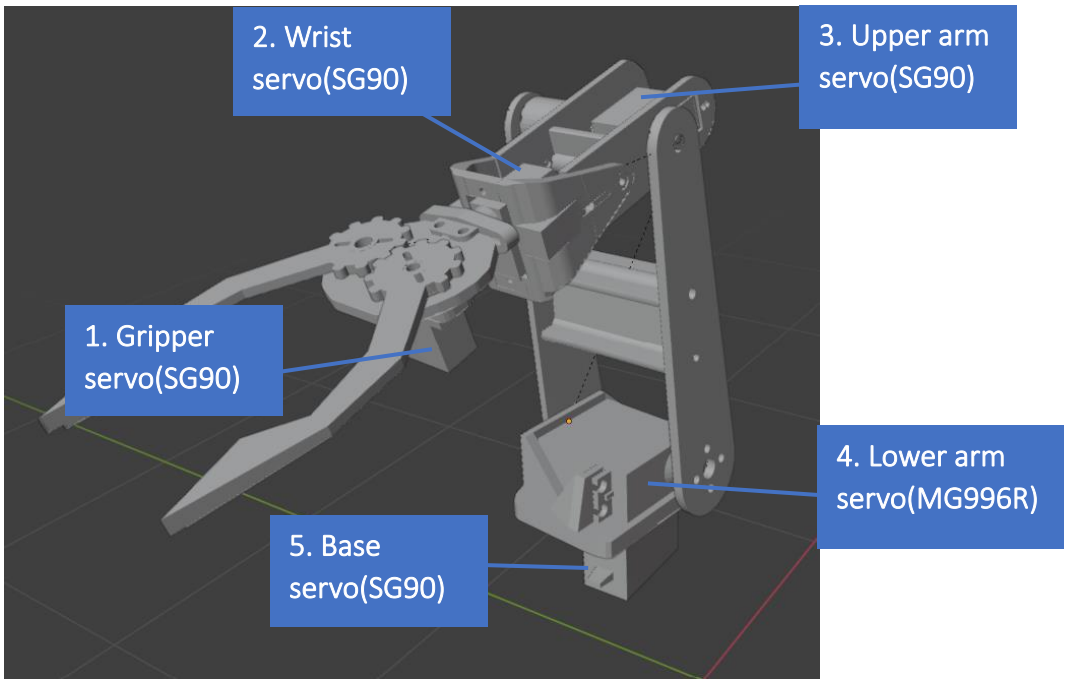

Figure 10: Servo placement of the robot arm

3.2.1 3D models of each part

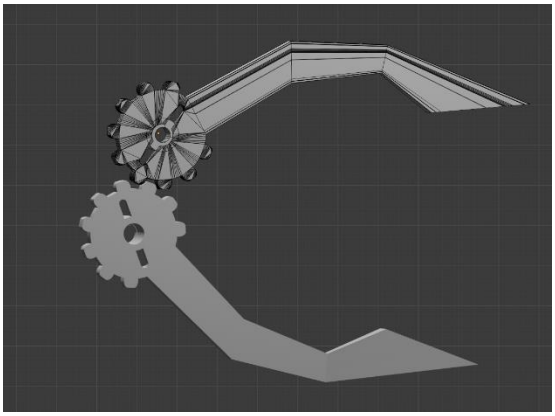

Figure 12: 3D model of the Gripper

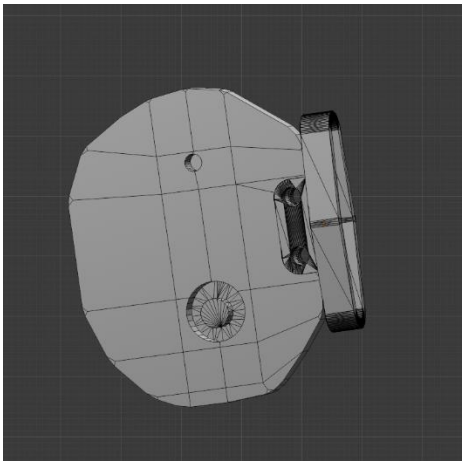

Figure 11: 3D model of the wrist

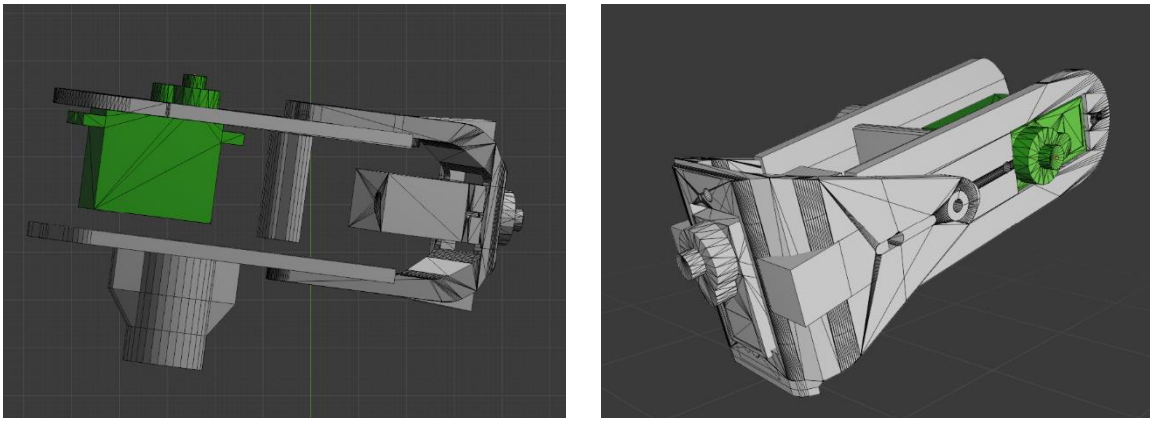

Figure 13: 3D model of the Upper arm (including servos)

Figure 14: 3D model of the Lower arm

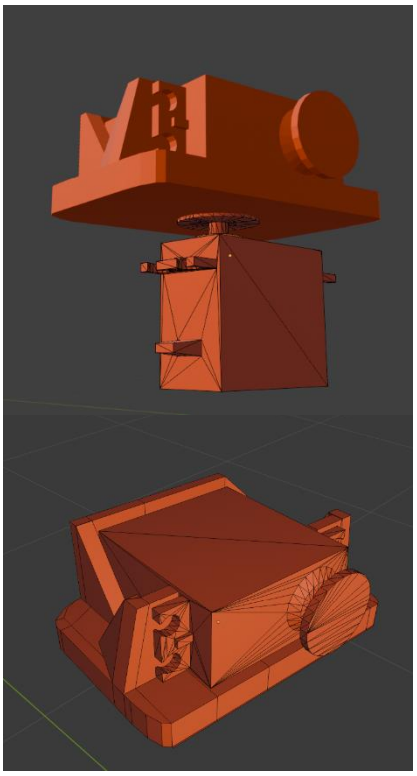

Figure 15: 3D model of the Base (Top image is with both MG996R servos)

3.3 PCB design

PCB is designed using EasyEDA standard online free PCB and circuit design software. PCB design for robot arm is shown below.

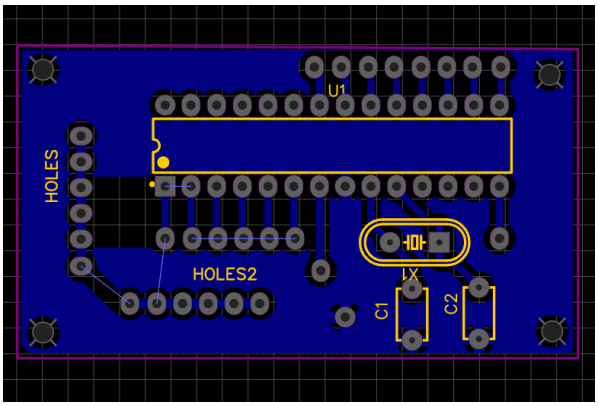

Figure 17: Top view of the PCB

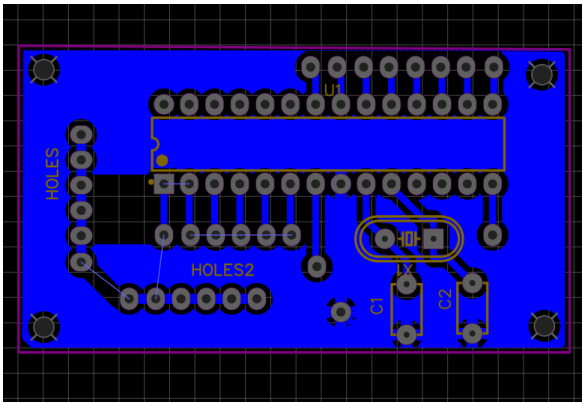

Figure 16: Bottom view of the PCB

4.Discussion and Conclusion

Main discussion based on the results obtained by analyzing and using the hand tracking robot arm can be categorized to 4 main types and complete review is as follows,

1. Accuracy and Responsiveness of Hand Tracking

- Tracking Accuracy: The MediaPipe framework was effective in recognizing and tracking hand landmarks, providing detailed and reliable coordinates of 21 hand points in real time. For most scenarios, the system detected hand movements with high accuracy, allowing for near-natural mimicry by the robotic arm. However, some minor inconsistencies were observed, especially with rapid hand movements or when certain landmarks (e.g., fingertips) were obscured in minor amount. But the overall tracking performance remained generally stable under moderate lighting as well as though lower lighting conditions. Usage of filtration techniques enhanced the smoothness of hand tracking for every landmark IDs used.
- Latency Analysis: The time delay from gesture detection in the Python program to the robotic arm's response is a critical factor in the system's performance. Main latency arises largely due to Bluetooth transmission speed and the time taken by the servos to reach the specified angles. This latency was not even noticeable (less than 20ms) for both slower movements and for fast gestures, indicating strong pillar for improvement if real-time high reactiveness.

2. Servo Movement and Precision

- Servo Positioning and Precision: The five servos in the robotic arm were calibrated to follow the angles of key hand joints (wrist and fingertips etc.) For most hand positions, the servos reached their designated angles within a margin of 2–3 degrees, demonstrating satisfactory precision. However, due to the limited torque of the SG90 and MG996R servos, slight deviations were observed under heavier loads or when trying to replicate extreme hand postures (most noticeable overload instance observed for lower arm part of the arm rig by a SG90 servo).
- Smoothness of Motion: The robotic arm’s movements were mostly smooth when controlled at moderate speeds. However, minor jitter was sometimes present, particularly with the SG90 servos, which have limited power and stability compared to higher-grade servos. This jitter could be attributed to rapid changes in position data transmitted from the hand-tracking program. Implementing a low-pass filter or smoothing algorithm in future iterations could improve the fluidity of the arm’s motion.

3. Wireless Communication Performance

- **Bluetooth Stability**: The HC-05 Bluetooth module provided stable communication within a range of up to 10 meters under unobstructed conditions, which met the range requirements for typical indoor applications. However, occasional data packet losses were observed, particularly when other Bluetooth devices were active nearby, causing slight delays in the arm's response. The module performed reliably for close-range communication, but environments with high interference could benefit from stronger or more advanced Bluetooth modules, such as Bluetooth 5.0.

4. System Power Consumption and Efficiency

- **Battery Life and Power Usage**: The system's power consumption primarily depends on the servo motors, the ATmega328P, and the Bluetooth module. With continuous use, the servos consumed significant power, especially the MG996R servos, which have a higher stall current. The ATmega328P, however, is efficient and consumed minimal power, even during active processing. Running the entire setup on 3 3.7V 18650 Li-ion batteries provided a considerable usage time.

Using the above key points, we can build up the strengths and limitations of the robot arm and this clears the way to improve its abilities for limitless usage over several fields. One of the primary strengths of this project was its success in effectively mimicking human hand movements through a robotic arm. The use of five servos enabled a realistic range of motion, especially for wrist and finger joints, allowing the robot arm to follow even complex gestures with reasonable accuracy. The project also leveraged MediaPipe advanced hand-tracking capabilities, which reliably provided detailed landmark data from a standard webcam. This approach bypassed the need for intricate sensors on the user's hand, resulting in a cost-effective solution without compromising on accuracy. Additionally, the Bluetooth connection enabled convenient wireless control, eliminating restrictive cables, and granting greater user mobility is a key advantage in many application scenarios.

However, several limitations were also noted during the project's implementation. The servos (SG90 and MG996R) used in the arm had torque and speed limitations, impacting performance, particularly during rapid movements or when operating under some loads. This constraint sometimes prevented the arm from maintaining smooth, responsive motion, which limited its ability to keep up with fast gestures. Lastly, the hand-tracking algorithm's sensitivity to lighting conditions and visibility posed some challenges; low light or occlusions affected the accuracy of tracking, while occasional Bluetooth interference impacted data transmission stability. Together, these limitations highlight areas where enhancements could significantly improve the overall performance and reliability of the system.

5.Applications and Future Development

5.1 Applications

The hand-tracking robotic arm developed in this project opens a range of possibilities across various fields. In industrial automation, it can enable operators to control machinery remotely in hazardous environments, reducing direct physical exposure to potential dangers. This remote control could even extend to managing multiple robotic systems simultaneously, allowing a single operator to perform several tasks in parallel, increasing productivity and operational efficiency.

In the medical field, this technology holds potential for telemedicine and robotic-assisted surgeries, where surgeons could control multiple surgical tools or robotic arms in unison, enhancing precision and flexibility in complex procedures. Gesture-based robotic arms could also be valuable in assistive technology, enabling individuals with physical disabilities to perform daily tasks independently. By programming the system to recognize a range of hand gestures, users could seamlessly control multiple devices within their homes, from opening doors to adjusting lights and managing home appliances, creating a smart, accessible environment tailored to their needs.

In military and defense applications, hand-tracking systems could support operations that require remote handling of objects, such as bomb disposal or tactical reconnaissance. With the capability to control multiple robotic systems at once, personnel could manage several robotic units in real time, allowing for efficient and coordinated field operations while maintaining a safe distance from hazardous zones.

Additionally, the system has applications in virtual and augmented reality (VR/AR), where it could be used to create highly immersive, gesture-based control schemes. Users could control robotic avatars or objects in a virtual space, providing realistic, synchronized physical responses. This could be beneficial not only for entertainment but also for VR-based training simulations, enabling users to operate virtual machinery or equipment using natural hand movements.

Lastly, in education and research, this project serves as a valuable tool for learning and experimentation in embedded systems, robotics, and computer vision. Students and researchers can use the setup to explore advanced concepts in human-computer interaction, multi-object control, and gesture recognition. The hand-tracking system's ability to control multiple devices simultaneously could encourage further development in distributed robotics, collaborative systems, and automated process control, making it a significant resource for advancing both theoretical and applied knowledge in engineering and technology.

Future Development

Building on the strengths of this project, several areas of future improvement and expansion are possible. First, enhancing the robot arm's motor setup with higher-torque, faster servos would enable it to perform rapid, complex movements more efficiently, extending its capability for high-load operations. Adding feedback sensors to the servos could provide real-time angle and torque data, allowing for more precise, adaptive control. Upgrading from Bluetooth to a higher-bandwidth communication protocol, such as Wi-Fi, could further reduce latency, especially in environments where Bluetooth interference may impact performance. On the software side, implementing machine learning to predict hand movements could enable the system to anticipate actions, improving response times. Finally, developing a mobile or web-based interface for remote control could enhance accessibility, allowing users to operate the arm from greater distances with ease. These advancements would further enhance the robotic arm's responsiveness, accuracy, and adaptability, opening new possibilities for real-world applications.

In conclusion, the hand-tracking robotic arm demonstrates a robust and effective design that successfully combines embedded systems, real-time data transmission, and advanced hand-tracking capabilities. Through extended software PWM outputs, the robot arm can achieve precise control over multiple servos simultaneously, enabling smooth, coordinated motion that mirrors human hand gestures. Fast data transmission via Bluetooth allows for responsive control, while the MediaPipe library based hand-tracking ensures quick and accurate capture of hand landmarks. This system has proven stable and reliable, even when performing complex gestures, making it a versatile tool with applications in fields ranging from industrial automation to assistive technology. Overall, the project highlights the potential of affordable, embedded microcontroller systems to deliver advanced functionality and high performance, demonstrating an impressive balance of speed, precision, and stability in a real-time robotic application.

6.Appendix

6.1 Python code for hand tracking

```
import cv2
import mediapipe as mp
import serial
import time
import numpy as np

previous_ema = None
def smoothyy(data_point, previous_ema=None, smoothing_factor=0.1):

    if previous_ema is None:
        # Initialize EMA to the first data point if no previous EMA exists
        previous_ema = data_point
    else:
        # Update the EMA based on the new data point
        new_avg = (smoothing_factor * data_point) + (1 - smoothing_factor) * previous_ema

        if data_point > previous_ema + 50:
            new_avg = data_point
        elif data_point < previous_ema - 50:
            new_avg = data_point

        previous_ema = new_avg

    return int(previous_ema)

previous_ema1 = None
def smoothyy1(data_point, previous_ema1=None, smoothing_factor=0.1):

    if previous_ema1 is None:
        # Initialize EMA to the first data point if no previous EMA exists
        previous_ema1 = data_point
    else:
        # Update the EMA based on the new data point
        new_avg = (smoothing_factor * data_point) + (1 - smoothing_factor) * previous_ema1

        if data_point > previous_ema1 + 80:
            new_avg = data_point
        if data_point < previous_ema1 - 80:
            new_avg = data_point

        previous_ema1 = new_avg

    return int(previous_ema1)

previous_ema2 = None
def smoothyy2(data_point, previous_ema2=None, smoothing_factor=0.1):

    if previous_ema2 is None:
        # Initialize EMA to the first data point if no previous EMA exists
```

```
        previous_ema2 = data_point
    else:
        # Update the EMA based on the new data point
        new_avg = (smoothing_factor * data_point) + (1 - smoothing_factor) * previous_ema2

        if data_point > previous_ema2 + 50:
            new_avg = data_point
        elif data_point < previous_ema2 - 50:
            new_avg = data_point

        previous_ema2 = new_avg

    return int(previous_ema2)

previous_ema3 = None
def smoothyy3(data_point, previous_ema3=None, smoothing_factor=0.1):

    if previous_ema3 is None:
        # Initialize EMA to the first data point if no previous EMA exists
        previous_ema3 = data_point
    else:
        # Update the EMA based on the new data point
        new_avg = (smoothing_factor * data_point) + (1 - smoothing_factor) * previous_ema3

        if data_point > previous_ema3 + 80:
            new_avg = data_point
        if data_point < previous_ema3 - 80:
            new_avg = data_point

        previous_ema3 = new_avg

    return int(previous_ema3)

previous_ema4 = None
def smoothyy4(data_point, previous_ema4=None, smoothing_factor=0.1):

    if previous_ema4 is None:
        # Initialize EMA to the first data point if no previous EMA exists
        previous_ema4 = data_point
    else:
        # Update the EMA based on the new data point
        new_avg = (smoothing_factor * data_point) + (1 - smoothing_factor) * previous_ema4

        if data_point > previous_ema4 + 50:
            new_avg = data_point
        elif data_point < previous_ema4 - 50:
            new_avg = data_point

        previous_ema4 = new_avg

    return int(previous_ema4)

send_data = False #used to ensure that coordinates are sending to robot arm
Connected_arm = False #used to ensure that bluetooth connected

def map_value(val, in_min, in_max, out_min, out_max): #map function foe servo angles
    return (val - in_min) * (out_max - out_min) // (in_max - in_min) + out_min
```

```
#Setup window sizes
servo_window_width = 640  #Width of the servo angles display (same as hand tracking)
servo_window_height = 480  #Height of the servo angles display (same as hand tracking)

# Initialize MediaPipe Hands
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(static_image_mode=False,
                       max_num_hands=1,
                       min_detection_confidence=0.5,
                       min_tracking_confidence=0.5)

# Initialize MediaPipe Drawing
mp_drawing = mp.solutions.drawing_utils

# Open the webcam
cap = cv2.VideoCapture(0)

try:
    atm = serial.Serial('COM12 ', 9600)
    Connected_arm = True
except serial.SerialException as e:
    print(f"Error: {e}")
    cap.release()
    cv2.destroyAllWindows()
    exit()

alpha = 0.8  # Smoothing factor
ema_y = None  # Initialize with None

def map_range(value, from_min, from_max, to_min, to_max):
    """ Map a value from one range to another. """
    if value<50:
        value=50
    if value>200:
        value= 200

    return int((value - from_min) * (to_max - to_min) / (from_max - from_min) + to_min)

def map_range2(value, from_min, from_max, to_min, to_max):
    """ Map a value from one range to another. """
    if value>500:
        value= 500
    if value<200:
        value= 200

    return int((value - from_min) * (to_max - to_min) /
               (from_max - from_min) + to_min)

# Define the red border region (e.g., 50 pixels from each side)
border_thickness = 50
data = "0,0,0,0,0\n"

while cap.isOpened():
    success, image = cap.read()
    if not success:
        print("Failed to capture image")
        break
```

```
    # Flip the image horizontally for a later selfie-view display
    image = cv2.flip(image, 1)

    # Convert the BGR image to RGB
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Process the image and find hands
    results = hands.process(image_rgb)

    h, w, c = image.shape
    #print(w)

    # Draw the red border
    cv2.rectangle(image, (border_thickness, border_thickness),
                  (w - border_thickness, h - border_thickness), (0, 255, 0), thickness=2)
    #cv2.putText(image, "Keep the hand within the safe zone",
                                #(50, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 20), 1)
    # Define the text and position
    text = "Keep your hand inside the Green Box"
    position = (50, 450)
    font = cv2.FONT_HERSHEY_SIMPLEX
    scale = 0.5
    color = (255, 255, 255)  # White color for the text
    thickness = 1

# Calculate the size of the text box
    (text_width, text_height), baseline = cv2.getTextSize(text, font, scale, thickness)
    x, y = position
# Define the box position and size
    box_coords = ((x, y + baseline), (x + text_width, y - text_height - baseline))

# Draw the filled rectangle (box) with black color
    cv2.rectangle(image, box_coords[0], box_coords[1], (0, 0, 0), cv2.FILLED)

# Put the white text on top of the black box
    cv2.putText(image, text, (x, y), font, scale, color, thickness)

    #data =[]
    # Draw hand landmarks and track index finger and thumb tips
    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(image, hand_landmarks, mp_hands.HAND_CONNECTIONS)

            # Track index finger tip (landmark 8) and thumb tip (landmark 4)
            lm_thumb = hand_landmarks.landmark[4]
            lm_index = hand_landmarks.landmark[8]

            lm_middle = hand_landmarks.landmark[12]
            lm_middle_base = hand_landmarks.landmark[9]

            lm_ring = hand_landmarks.landmark[16]
            lm_ring_base = hand_landmarks.landmark[13]

            lm_small = hand_landmarks.landmark[20]
            lm_small_base = hand_landmarks.landmark[17]

            palm_base = hand_landmarks.landmark[0] # base
```

```
h, w, c = image.shape
            cy_index, cy_thumb, cx_index, cx_thumb= int(lm_index.y * h), int(lm_thumb.y * h),
int(lm_index.x * w), int(lm_thumb.x * w)
            cy_middle, cy_middle_base, cx_middle, cx_middle_base= int(lm_middle.y * h),
int(lm_middle_base.y * h), int(lm_middle.x * w), int(lm_middle_base.x * w)
            cy_ring, cy_ring_base, cx_ring, cx_ring_base= int(lm_ring.y * h), int(lm_ring_base.y *
h), int(lm_ring.x * w), int(lm_ring_base.x * w)
            cy_small, cy_small_base, cx_small, cx_small_base= int(lm_small.y * h),
int(lm_small_base.y * h), int(lm_small.x * w), int(lm_small_base.x * w)

            cy_base= int(palm_base.y*h)
            cx_base= int(palm_base.x*w)

            #scale constant
            #using base and middle finger base
            x0 = (cx_middle_base-cx_base)** 2
            y0 = (cy_middle_base-cy_base)** 2
            main_diff = int(np.sqrt(x0 + y0))
            centered_diff = abs(main_diff-150) #150 is the centered scale factor..... 150
is crucial
            #main_diff = scale_y1(main_diff)

            #gripping //.....100 degrees
            x1 = (cx_index-cx_thumb)**2
            y1 = (cy_index-cy_thumb)**2
            grip_diff = int((np.sqrt(x1 + y1)) + centered_diff) #.....


            #wrist rotating.....by small finger.....// 90 degrees
            x2 = (cx_small-cx_base)**2
            y2 = (cy_small-cy_base)**2
            small_diff = int(np.sqrt(x2 + y2)) + centered_diff

            #elbow rotating.....by ring finger.....// 90 degrees
            x3= (cx_ring-cx_ring_base)**2
            y3 = (cy_ring-cy_ring_base)**2
            ring_diff = int(np.sqrt(x3 + y3)) + centered_diff

            #shoulder rotating.....by middle finge.....// 90 degrees
            x4= (cx_middle-cx_middle_base)**2
            y4 = (cy_middle-cy_middle_base)**2
            middle_diff = int(np.sqrt(x4 + y4)) + centered_diff

            gripdiff_map =(map_range(grip_diff, 50, 200, 2000, 0)) #1920 is the finest gipper
value i set
            #gripp= smoothyy(gripdiff_map,previous_ema)
            previous_ema = smoothyy(gripdiff_map, previous_ema, smoothing_factor=0.1)

            smalldiff_mapp = (map_range2(cy_base, 200, 500, 0, 2000 ))
            previous_ema1 = smoothyy1(smalldiff_mapp, previous_ema1, smoothing_factor=0.1)

            ringdiff_mapp = (map_range(ring_diff, 50, 200, 200, 2000 ))
            previous_ema2 = smoothyy2(ringdiff_mapp, previous_ema2, smoothing_factor=0.1)#......

            middlediff_map = (map_range(middle_diff, 50, 200, 2000, 0 ))
```

```
            previous_ema3 = smoothyy3(middlediff_map, previous_ema3, smoothing_factor=0.1)#......

            cxbase_mapp = (map_range2(cx_base, 200, 500, 0, 2000 ))
            previous_ema4 = smoothyy4(cxbase_mapp, previous_ema4, smoothing_factor=0.1)

            #data = f"{middlediff_map},{middle_diff}\n"
            #data =
f"{previous_ema},{previous_ema1},{previous_ema2},{previous_ema3},{previous_ema4}\n"
            #data = f"{thumb_rel},{thumb_relp},{thumb_rel2}\n"
            #data = f"{di}\n"

            out_of_bounds = False  # Flag to check if any part of the hand is out of the box

            for landmark in [lm_thumb, lm_index, lm_middle, lm_ring, lm_small,palm_base]:
                cx, cy = int(landmark.x * w), int(landmark.y * h)
                if (cx < border_thickness or cx > w - border_thickness or cy < border_thickness or
cy > h - border_thickness):
                    out_of_bounds = True
            # Draw the warning text and the red border
                    cv2.putText(image, "Get your hand back to the box",
                    (80, 240), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 250), 1)
                    cv2.rectangle(image, (border_thickness, border_thickness),
                    (w - border_thickness, h - border_thickness), (0, 0, 255), thickness=4)

            if out_of_bounds:
                data = f"{1000},{1000},{1000},{1000},{1000}\n"
            else:
                data =
f"{previous_ema},{previous_ema1},{previous_ema2},{previous_ema3},{previous_ema4}\n"
                #data = f"{gripdiff_map},{previous_ema},{previous_ema2},{previous_ema3}\n"
            #print(data)

            try:
                atm.write(data.encode())
                send_data = True
            except serial.SerialException as e:
                print(f"Serial error: {e}")
                break
##
    #Arm dsiplay window
    servo_display = cv2.imread('arm12.png')
    data_values = data.strip().split(',')

    #Convert the string values to integers or floats
    data_values = [int(value) for value in data_values]
    for i, angle in enumerate(data_values):
        if i==0:
            mapped_angle = map_value(data_values[0], 0, 2000, 90, 0)
            text = f"Angle: {mapped_angle}"
            cv2.putText(servo_display, text, (200, 267), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255,
0), 1)
        if i==1:
            mapped_angle = map_value(data_values[1], 0, 2000, 0, 90)
            text = f"Angle: {mapped_angle}"
            cv2.putText(servo_display, text, (289, 53), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255,
0), 1)
```

```
            mapped_angle = map_value(data_values[2], 0, 2000, 90, 0)
            text = f"Angle: {mapped_angle}"
            cv2.putText(servo_display, text, (520, 79), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0),
1)
        if i==3:
            mapped_angle = map_value(data_values[3], 0, 2000, 0, 90)
            text = f"Angle: {mapped_angle}"
            cv2.putText(servo_display, text, (300, 308), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 165,
255), 1)
        if i==4:
            mapped_angle = map_value(data_values[4], 0, 2000, 0, 90)
            text = f"Angle: {mapped_angle}"
            cv2.putText(servo_display, text, (440, 428), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 165,
255), 1)

    if Connected_arm:
        cv2.putText(servo_display, "Bluetooth Connected", (51, 400), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
(0, 250, 0), 1)
    else:
        cv2.putText(servo_display, "Bluetooth Disonnected", (51, 410), cv2.FONT_HERSHEY_SIMPLEX,
0.5, (0, 0, 255), 1)

    if send_data:
        cv2.putText(servo_display, "Coordinates Sending...", (51, 430), cv2.FONT_HERSHEY_SIMPLEX,
0.5, (255, 178, 102), 1)
    else:
        cv2.putText(servo_display, "Sending Failed", (51, 430), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,
0, 255), 1)

    # Create a combined display with the servo display on the left and hand tracking on the right
    image_height, image_width, _ = image.shape
    combined_display_width = image_width + servo_window_width  #Total width is 640 (Robot arm image)
+ 640 (Hand tracking image)

    # Create a blank image for the combined display (480 height and 1280 width)
    combined_display = np.zeros((image_height, combined_display_width, 3), dtype=np.uint8)

    # Place servo display on the left
    combined_display[:servo_window_height, :servo_window_width] = servo_display

    # Place hand tracking image on the right
    combined_display[:image_height, servo_window_width:servo_window_width + image_width] = image

    # Show the combined display
    cv2.imshow('Tracking Window', combined_display)

    # Break the loop on 'q' key press
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

    # Optional: Small delay to manage data rate
    #time.sleep(0.02)  # 20ms delay

# Release the capture and close the windows
cap.release()
cv2.destroyAllWindows()
atm.close()
```

6.2 C code for ATmega328p micro-controller

```
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <avr/interrupt.h>

#define F_CPU 16000000UL
#define BAUD 9600
#define MYUBRR F_CPU/16/BAUD-1

#define NUM_CHANNELS 5  //Number of channels (OCR1A values)
#define PIN_HIGH(port, pin) (port |= (1 << pin))  //Set pin high
#define PIN_LOW(port, pin) (port &= ~(1 << pin))  //Set pin low

volatile uint16_t array1[NUM_CHANNELS] = {1999, 1999, 1999, 1999,
1999};  //default array to store pulse widths for each servos(5 servos)
volatile uint16_t array12[5];

volatile uint8_t currentChannel = 0;  //Current channel index

//volatile uint16_t elapsedTime = 0;
 //uint16_t a = 0;

void USART_Init(unsigned int ubrr);
void USART_Transmit(unsigned char data);
unsigned char USART_Receive(void);

void init_timer1_servo(void);
void set_pulse_width(uint16_t pulse_width);

ISR(TIMER1_COMPA_vect) {
    PIN_LOW(PORTB, currentChannel);  //setting the current pin low when OCR1A is
reached
}

ISR(TIMER1_OVF_vect) {

    //updating the next channel
    currentChannel++;
    if (currentChannel >= NUM_CHANNELS) {
        currentChannel = 0;  //first channel
    }
```

```
    set_pulse_width(array1[currentChannel]);
    PIN_HIGH(PORTB, currentChannel);  //set the new pin high at the start of the
cycle

}

// ISR(TIMER0_OVF_vect){
//   elapsedTime++;
// }

int main(void) {
    USART_Init(MYUBRR);
    init_timer1_servo();

    TCCR0B|=(1<<CS00);
    TIMSK0|=(1<<TOIE0);
    TIMSK1 |= (1 << OCIE1A) | (1 << TOIE1) | (1 << OCIE1B);
    sei();

    char buffer[500];  //buffer to hold the received string-->data bit size is
about 250 bits, 500 is enough then
    uint8_t index = 0;
    uint8_t count = 0;  //To count the number of numbers processed
    //array to store updated coordinates
    int  numbers[5];
    //buffer flushing
    memset(buffer, 0, sizeof(buffer));

    while (1) {

        char received = USART_Receive();

        if (received == ',' || received == '\n') {
            buffer[index] = '\0';  //Null-terminate the string

            if (count < 5) {
                numbers[count++] = atoi(buffer); //Assigning and converting the
string to an integer and store it in the numbers array
            }

            index = 0;  //reset the buffer index for the next number

            //If a newline character is received, process the numbers
            if (received == '\n' && count == 5) {
```

```
            if (numbers[0] >=0 && numbers[0] <= 2000) { //upto 3000 we can tilt
90 degrees
                cli();
                array1[0]=numbers[0]+1999; //array[0] is PB1 // for
grip.................//about 110 degrees (sg90)
                sei();
              }
              if  (numbers[1] >=0 && numbers[1] <= 2000) {
                cli();
                array1[1]=numbers[1]+1999;//array[1] is PB2 //for
wrist............//about 90 degrees (sg90)
                sei();
              }
               if (numbers[2] >=0 && numbers[2] <= 2000) {
                cli();
                array1[2]=numbers[2]+1999;//array[2] is PB3 //for
elbow............//about 90 degrees (sg90)
                sei();
              }
              if  (numbers[3] >=0 && numbers[3] <= 2000) {
                cli();
                array1[3]=numbers[3]+1999;//array[3] is PB4  //for
shoulder.........//about 90 degrees (MG 995).......
                sei();
              }
               if (numbers[4] >=200 && numbers[4] <= 2000) {
                cli();
                array1[4]=numbers[4]+1999;//array[4] is PB0 //for base
rotating.......//about 180 degrees (MG995)....
                sei();
              }

              count = 0;  //reset count for the next set of 5 numbers
          }
      } else {
          buffer[index++] = received;  //store the character in the
buffer
      }

      USART_Transmit(received);
  }
}
//.....................................................................
....
void USART_Init(unsigned int ubrr) {
```

```
    UBRR0H = (unsigned char)(ubrr >> 8);
    UBRR0L = (unsigned char)ubrr;
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
}

void USART_Transmit(unsigned char data) {
    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = data;
}

unsigned char USART_Receive(void) {
    while (!(UCSR0A & (1 << RXC0)));
    return UDR0;
}
void init_timer1_servo() {

    //Timer1 to Fast PWM modewith ICR1 as top value
    TCCR1A |= (1 << WGM11);
    TCCR1B |= (1 << WGM12) | (1 << WGM13) | (1 << CS11); //Prescaler 8

    ICR1= 7999; //Customized value here....

    DDRB |= 0x1F;  //Set PB0 to PB4 as output(5 channels)

    OCR1A = array1[0] ;

}

void set_pulse_width(uint16_t pulse_width) {
    OCR1A = pulse_width;  //updating the OCR1A using pulse_width value
}
```

7.References

GitHub. (2024). *google-ai-edge/mediapipe*. [online] Available at: https://github.com/google-ai-edge/mediapipe.

Components101 (2017). *Servo Motor SG-90 Basics, Pinout, Wire Description, Datasheet*. [online] Components101. Available at: https://components101.com/motors/servo-motor-basics-pinout-datasheet.

Components101 (2019). *MG996R Servo Motor Datasheet, Wiring Diagram & Features*. [online] components101.com. Available at: https://components101.com/motors/mg996r-servo-motor-datasheet.

Google for Developers. (n.d.). *MediaPipe Solutions guide | Edge*. [online] Available at: https://ai.google.dev/edge/mediapipe/solutions/guide.

Google for Developers. (n.d.). *Hand landmarks detection guide | Edge*. [online] Available at: https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker.

Boesch, G. (2023). *MediaPipe: Google's Open Source Framework for ML solutions (2023 Guide)*. [online] viso.ai. Available at: https://viso.ai/computer-vision/mediapipe/.

Arduino Forum (2018). *Making custom frequency PWM*. [online] Arduino Forum. Available at: https://forum.arduino.cc/t/making-custom-frequency-pwm/533174 [Accessed 1 Nov. 2024].

Arduino.cc. (2024). Available at: https://docs.arduino.cc/tutorials/generic/secrets-of-arduino-pwm/.

signal, P. (2020). *'Manually' generating a PWM signal*. [online] Arduino Stack Exchange. Available at: https://arduino.stackexchange.com/questions/73348/manually-generating-a-pwm-signal [Accessed 1 Nov. 2024].