



**Department of Electronic Engineering**

**Final Project Report**

**Title: Robotic Arm & MATLAB Kinematics Interface**

**Submitted By:**

Laiba Shehzad (2021-BEE-009)  
Aroosa Bibi (2021-BEE-003)  
Palwasha Bibi (2021-BEE-015)  
Ayesha Shabbir Mirza (2021-BEE-004)  
Shanza Noor (2021-BEE-016)  
Afshan Bibi (2021-BEE-001)  
Muneeba Bibi (2021-BEE-013)

**Submitted To:**

Engr. Haroon Waseem

**Date of Submission:**

12-June-2025

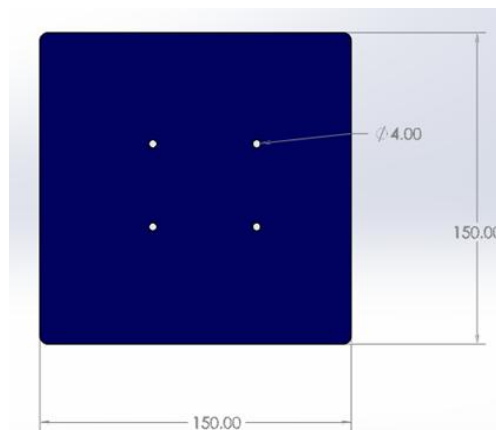
## Introduction:

The growing need for automation in industrial and domestic applications has resulted in the use of robotic systems in a wide range of applications. In this project, the design and implementation of a 4-DOF robotic arm with kinematic modeling in MATLAB are considered. Object manipulation and precise positioning capabilities (forward and inverse kinematics) are some of the functionalities of the robotic system that is controlled through an ESP32 microcontroller. Mechanical design is simulated with the CAD tools, whereas MATLAB App Designer is applied to create a comprehensive GUI enabling real-time control and simulation. The project will fill the gap between mechanical design and algorithmic control, and will be a scalable solution to automation problems such as pick and place operations, lab automation, and sorting systems.

## 4 DOF Robotic Arm CAD Model:

### Parts of a Robot:

#### 1: Base Plate:



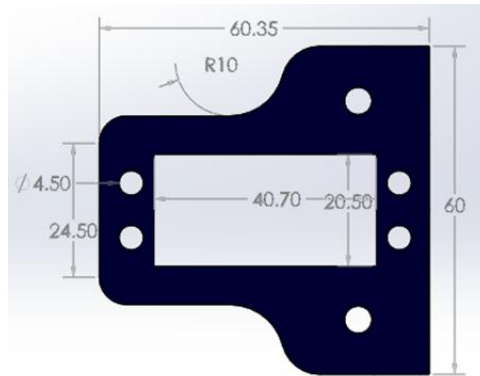
### Commands:

Rectangle, circle, and fillet are all included in the sketching section. After that, select the Extrude command.

### Working:

The bottom part of the robot is this part. It helps the robot stay put and remain fixed in one position. By balancing the robot's weight it keeps it from falling over.

## 2: Servo Mount:



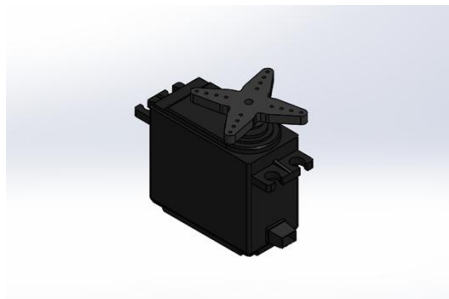
### Commands:

Rectangle, circle, and fillet are part of the sketch commands constructed in the Sketch by Designer Mode. After that, use the Extrude command.

### Working:

The special bracket exists to support and add strength to the servo motor. The joint also connects the first part of the robot arm to the base. Without it, the motor wouldn't stay in place and the link could not rotate as needed.

## 3: Servo Motor:



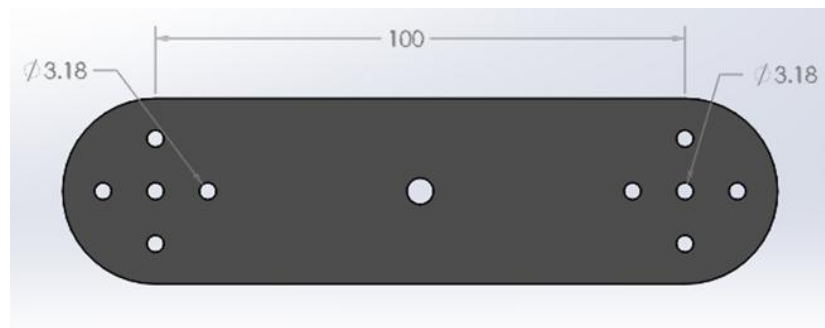
### Commands:

Rectangle, circle, and fillet are included in the set of sketching commands. You should then use the Box Extrude and Cut Extrude tools. Assembly of the servo is another task to complete.

### **Working:**

Movement in a robot is mainly created by a servo motor. This ensures that the robot's independent links move with each other. In CAD, we set the item's exact dimensions, made openings for the screws, and joined it with the other parts.

### **4: Link 1:**



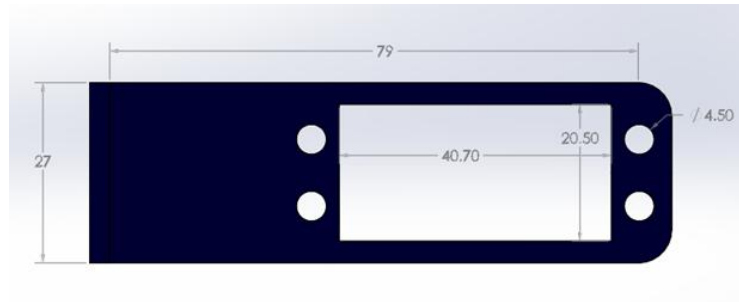
### **Commands:**

Rectangle, circle, and fillet are some of the commands used in sketching. After that, select the Extrude tool.

### **Working:**

This part is the first section of the robot's arm. It moves the original part to the next link like in human upper arm structure. It can be moved up and down thanks to the motor at the base.

### **5: Link 2:**



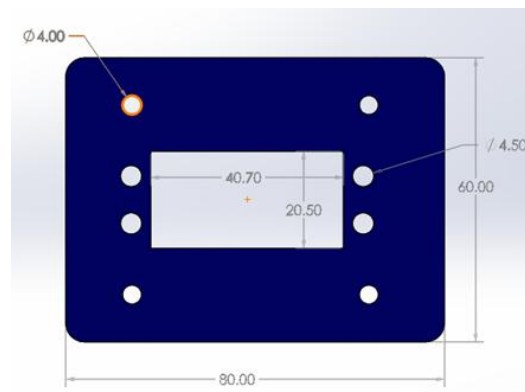
### Commands:

Sketching features have rectangles, lines, circles, and fillets. After that, select the Extrude command.

### Working:

The second piece in the upper limb is the arm, just like the forearm. It is joined to Link 1 and enhances how far the robot can reach. It can bend by using a flexible joint called an elbow.

## 6: Base Motor Mount:



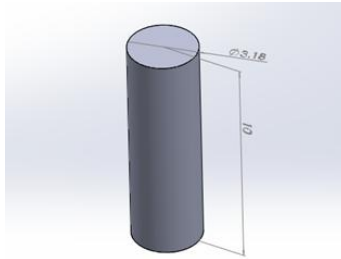
### Commands:

There are rectangle, circle, and fillet commands in the sketching menu. Then pick the Extrude command.

### Working:

That is where the motor sits that enables the arm to swing left to right (around the Z-axis). It serves as a turntable that controls all the robot's activities.

## 7: Pin



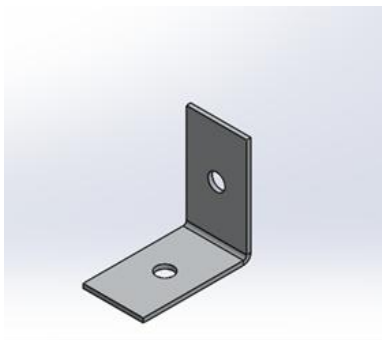
### Commands:

Sketching comes with a command for circles. After that, pick the button called Extrude.

### Working:

You can connect two moving parts, for instance links or joints, by using a pin. It rotates just like a door with a hinge.

## 8: Angle Bracket



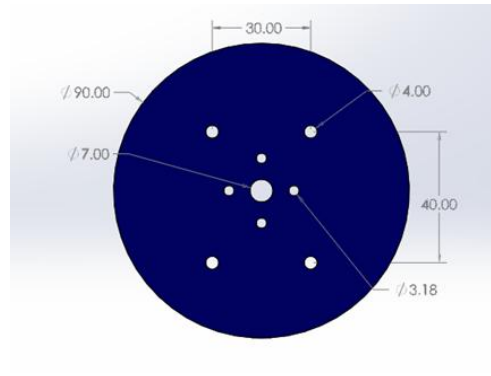
### Commands:

Rectangles, circles, and fillets are some of the tools included in sketching. Then you need to use Extrude Settings.

### Working:

This part helps join the base plate to the motor mount. It fixes the parts in the way that is required by the design.

### 9: Base Circular Plate:



#### Commands:

Referring to the menu, you will find that sketching commands are for creating rectangles, circles, and fillets. After that, I choose the Extrude command.

#### Working:

This part is also part of the base; it holds the rotating motor and robot arm. This gives extra strength and allows the robot arm to turn as well.

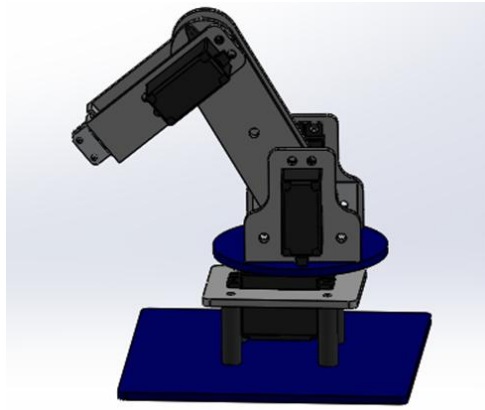
### 10: Gripper Assembly:



#### Working:

Grippers are at the robot's end, and they work similarly to a hand. Hand functions by opening and closing to move objects to different places.

### 11: Assembled Robot :



## **Commands:**

At assembly, the robot parts are imported, and then they are mated to become a 4 DOF robot.

The 3D model includes all elements needed for the robot arm: base, links, motors, and the gripper. Together they act as a complete working robot arm with 4 DOF.

## **Dimensions:**

### **All Revolute Joints & Link Breakdowns**

1. Joint 1 (base rotation)—about the Z-axis.
2. Joint 2 (Shoulder Joint)—raises or lowers the first arm segment.
3. Joint 3 (Elbow Joint)—controls bending between the gripper and lower arm.
4. Joint 4 (Gripper Actuation)—the servo opens and closes the gripper.

## **Links:**

- Link 1 – Base to Shoulder (100mm)
- Link 2 – Shoulder to Elbow (65 mm)
- Link 3 – Wrist to Gripper (0mm)



## General DH Table:

Then DH parameters will be like

	$a_{i-1}$	$\alpha_{i-1}$	$d_i$	$\theta_i$
1	0	90°	0	$\theta_1$
2	L1	0°	0	$\theta_2$
3	L2	0°	0	$\theta_3$
4	0	0°	0	$\theta_4$

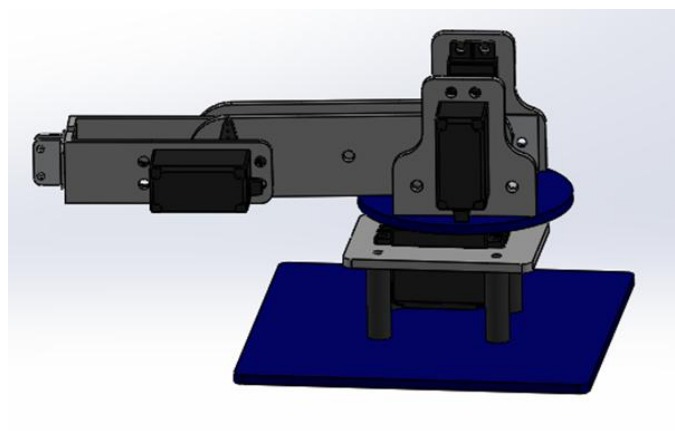
## Desired Pose:

Let's assume you want a pose like

- Arm extended straight forward in horizontal direction

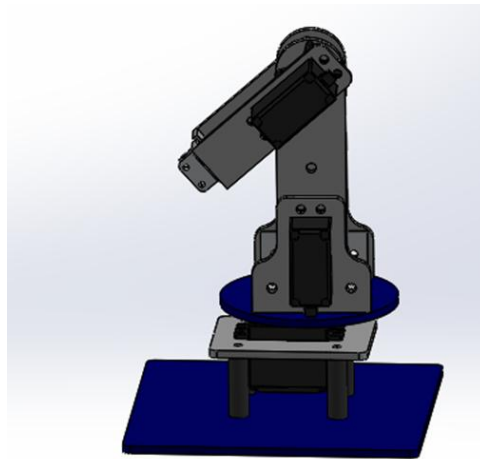
Then DH parameters will be like

	$a_{i-1}(\text{mm})$	$\alpha_{i-1}$	$d_i$	$\theta_i$
1	0	90°	0	0
2	100	0°	0	0
3	60	0°	0	0
4	0	0°	0	$\theta_4$



Or if you want the arm folded:

	$a_i(\text{mm})$	$\alpha_i$	$d_i$	$\theta_i$
1	0	90°	0	45
2	100	0°	0	-90
3	60	0°	0	0
4	0	0°	0	$\theta_4$



## Methodology

The whole project is divided into seven parts, the important one is the software and the other one is the hardware part.

## Hardware Development


### Mechanical (Kit) Assembly






First of all, the cutting of the design is done. Then, assembling parts of the robotic arm kit is done step by step, which can be seen in the video. The kit consists of acrylic sheets, nuts, spacers, etc.



## Selection of Electronic Components

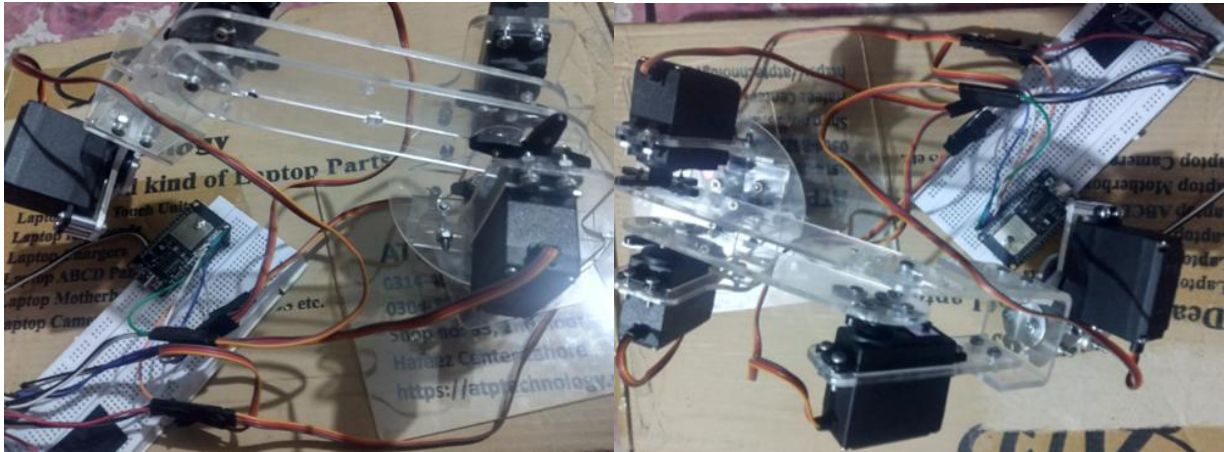
Electronic Components selection is the most important part of any project. The selection of Components depends on the desired application. The components are carefully chosen for the Self Robotic Arm Kit, and are according to the desired objectives of the project. The table of components below shows the component name, its picture, and gives a brief description. In this project, we used MG996R Servo motors because they can easily handle the robotic arm weight and payload, which is 8 grams. According to the given task. These motors can easily control the robot, providing accurate movement of joints.

Sr#	Components	Picture	Description	Quantity
1.	Esp32		It is a microcontroller that has two main features of Bluetooth and Wi-Fi	1

			connectivity	
2.	Servo motors MG996R		It has a torque of 11 kg/cm, and the angle of rotation is 0 to 180 degrees. For controlling joints, it is good enough and can carry loads up to 6 kg.	5
3.	Spacers		Spacers are used in the kit assembly of the arm for connecting the base to the upper part of the robot.	4
4.	Charger(Power Supply)		To give power to the circuit, an ESP32 5-volt Charger is used	1
5.	Gripper		For picking, holding, and releasing objects gripper is used, which can handle objects of different shapes. It gets power from servo motors.	1
6.	Miscellaneous Items		Wires, screwdrivers, etc	-

## Placement of Electronic Components

Servo motors are attached to the body or kit of the Robotic arm. One is attached at the base, the second and third are attached to the left and right links, the fourth is attached to



the joint, and the last is attached with a gripper for capturing things or objects.

ESP32 is placed on a breadboard with connections to the servo motors on the breadboard, and the power supply is connected to the breadboard with a common terminal of the servo motors so each motor gets power.

## Calibration of Servos

Servos are calibrated to get accurate results, as always, servos do not move in the range of 0 to 180. Arduino IDE is used here for the calibration code of servos. The calibration code is attached in the annex of the report. Servo motors are calibrated for their movement test to move from 0 to 180 degrees. Angles minimum and maximum angles of movement of servos are set for checking the movement area. It is calibrated by checking the movement in the safe area of the robot. Through this, we get the best range in which the servo moves without stopping and jerking. After calibration, I came to know that servo 5, which is attached to a gripper, has a range of 0 to 90. In this range, it can move safely. The angles of rotation of the servos are:

Servo-1: Angle of rotation is from 0 to 180 degrees

Servo-2: 30 to 180 degrees

Servo-3: 0 to 150 degrees

Servo-4: 50 to 180 degrees

Servo-5: 0 to 90 degrees

## Code

```
sketch_may21a.ino
40 // // RESET POSITIONS
41 // servo1.write(0);
42 // delay(500);
43 // servo2.write(0);
44 // delay(500);
45 // servo3.write(0);
46 // delay(500);
47 // servo4.write(0);
48 // delay(500);
49 // servo5.write(0);
50 // delay(500);
51
52 // delay(2000);
53 // servo2.write(120); // 150 to 0 // opposite with servo3
54 // delay(500);
55 // servo3.write(60); // 30 to 180 // opposite with servo2
56 // servo5.write(90); // gripper 90 to 0
57 servo4.write(120); // 50 to 180
58 }
```

Output

```
Writing at 0x0003edb0... (63 %)
Writing at 0x000445a5... (72 %)
Writing at 0x0004aeab... (81 %)
Writing at 0x00055123... (90 %)
Writing at 0x0005ae39... (100 %)
Wrote 308896 bytes (165126 compressed) at 0x00010000 in 2.6 seconds (effective 948.6 kbit/s)...
Hash of data verified.

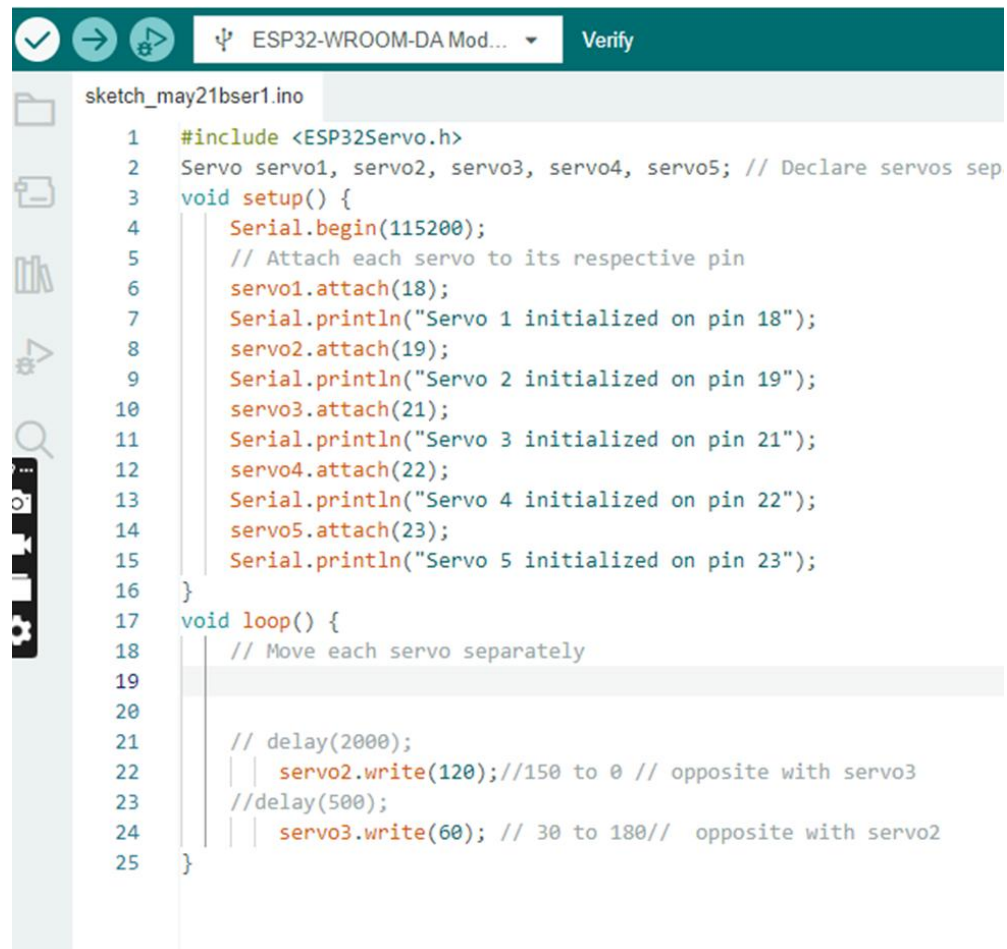
Leaving...
Hard resetting via RTS pin...
```

## BOM

BOM level	Part name	Description	Quantity	Cost
1	Acrylic sheet	For making kit of	-	2000+1300=3300

	cutting	robotic arm		
2	ESP 32	Microcontroller with Wi-Fi and Bluetooth	1	1300
3	Charger	For power supply charger of 5 volt	1	500
4	Servo Motors	For the movement of joints and controlling them accurately	1*5	1000*5=5000
5	Gripper	For picking up and gripping objects	1	1000
6	Jumper wires	For making electrical connections	-	100
7	Nuts and spacers	For assembling and connecting the parts of the robotic arm	-	500
			Total	11700.

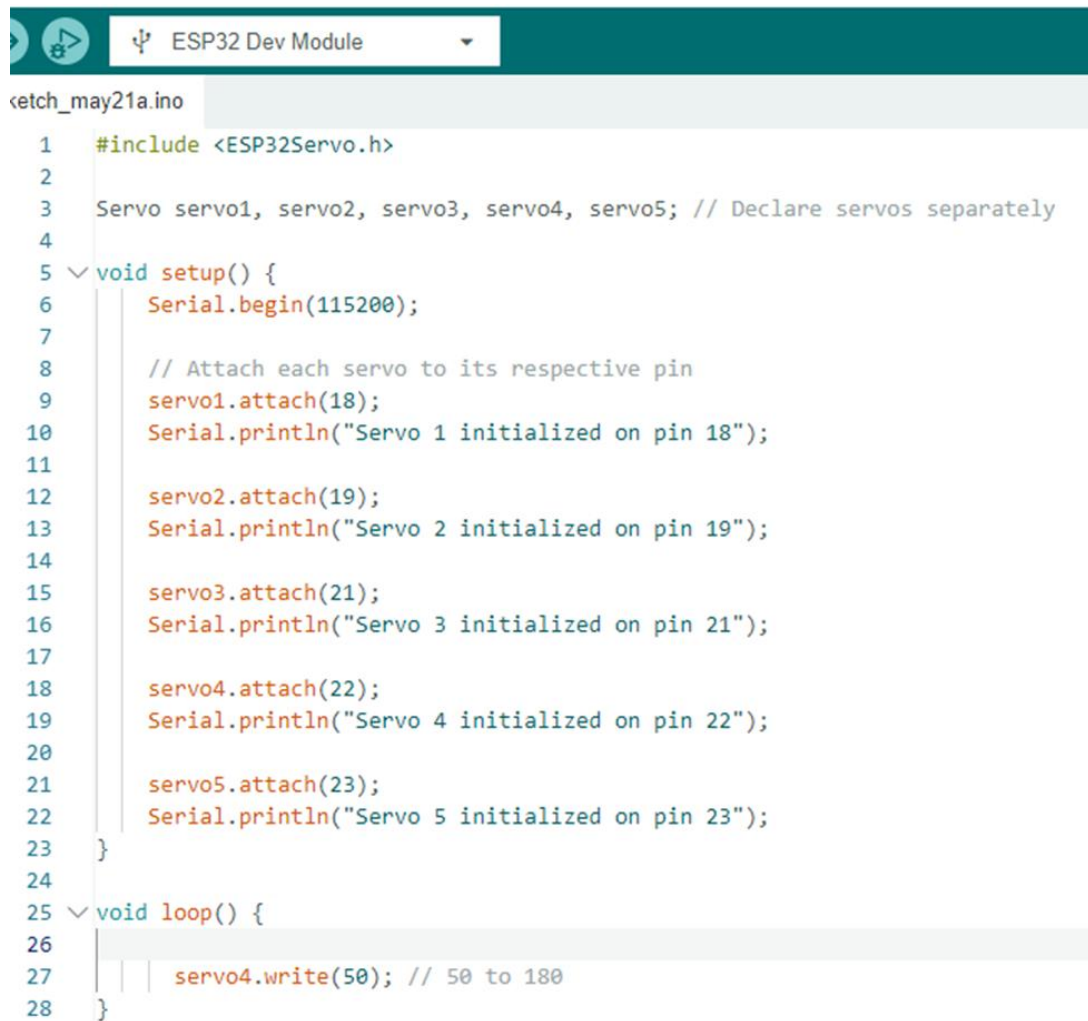
## Code for Calibration



The image shows the Arduino IDE interface. At the top, the board is set to 'ESP32-WROOM-DA Mod...' and the 'Verify' button is visible. The sketch file is named 'sketch\_may21bser1.ino'. The code defines five servos and initializes them in the setup function. The loop function contains a delay and writes to servo2 and servo3.

```
1  #include <ESP32Servo.h>
2  Servo servo1, servo2, servo3, servo4, servo5; // Declare servos sep
3  void setup() {
4      Serial.begin(115200);
5      // Attach each servo to its respective pin
6      servo1.attach(18);
7      Serial.println("Servo 1 initialized on pin 18");
8      servo2.attach(19);
9      Serial.println("Servo 2 initialized on pin 19");
10     servo3.attach(21);
11     Serial.println("Servo 3 initialized on pin 21");
12     servo4.attach(22);
13     Serial.println("Servo 4 initialized on pin 22");
14     servo5.attach(23);
15     Serial.println("Servo 5 initialized on pin 23");
16 }
17 void loop() {
18     // Move each servo separately
19
20
21     // delay(2000);
22     | | servo2.write(120); // 150 to 0 // opposite with servo3
23     // delay(500);
24     | | servo3.write(60); // 30 to 180 // opposite with servo2
25 }
```





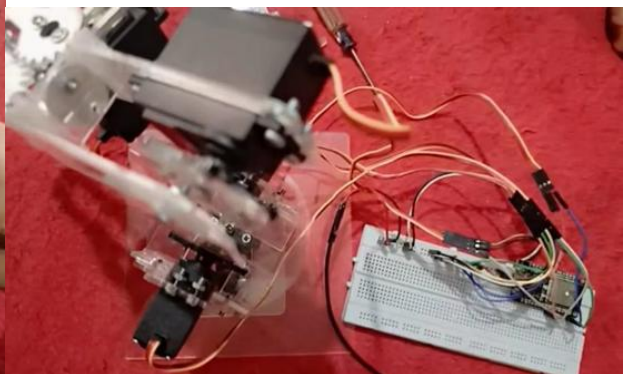
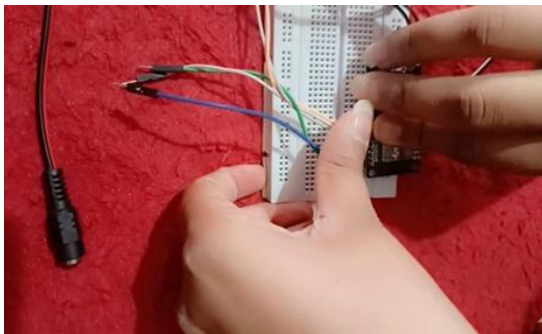
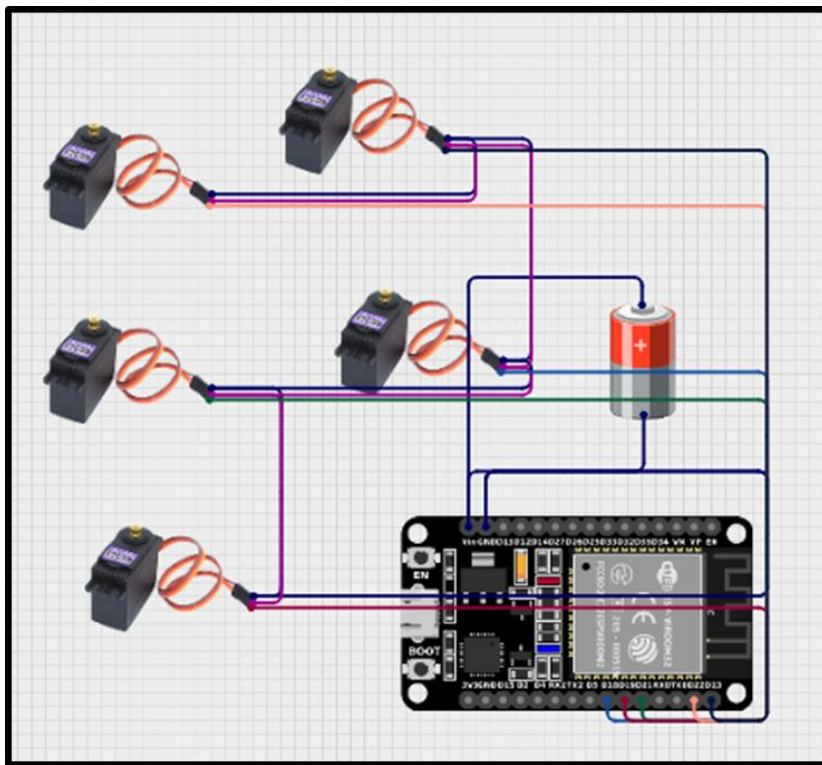
```
ESP32 Dev Module

catch_may21a.ino
1  #include <ESP32Servo.h>
2
3  Servo servo1, servo2, servo3, servo4, servo5; // Declare servos separately
4
5  void setup() {
6      Serial.begin(115200);
7
8      // Attach each servo to its respective pin
9      servo1.attach(18);
10     Serial.println("Servo 1 initialized on pin 18");
11
12     servo2.attach(19);
13     Serial.println("Servo 2 initialized on pin 19");
14
15     servo3.attach(21);
16     Serial.println("Servo 3 initialized on pin 21");
17
18     servo4.attach(22);
19     Serial.println("Servo 4 initialized on pin 22");
20
21     servo5.attach(23);
22     Serial.println("Servo 5 initialized on pin 23");
23 }
24
25 void loop() {
26
27     servo4.write(50); // 50 to 180
28 }
```

**Wiring**

**Diagram and Physical Assembly Lead**

**Circuit Diagram:**



This is the Circuit Diagram for the Robotic Arm. We use the app.cirkitdesigner for designing the circuit.

### **Purpose of the Components in the circuit:**

#### **ESP32:**

We use an ESP32 microcontroller that is 30 pin. It serves as the microcontroller in the circuit that manages the control of signals from the MG996R servos through the GPIO pins. It is responsible for processing inputs and then generating outputs for controlling the servo's position based on the programmed logic.

#### **Servos MG996R:**

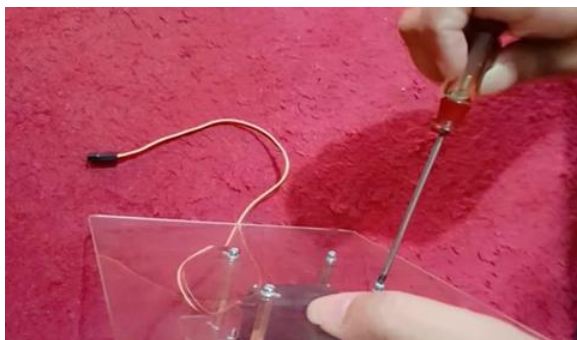
The Servos MG996R are high-torque and high-speed motors. These are commonly used in robotics and remote control projects. These are durable, reliable, and efficient for various applications.

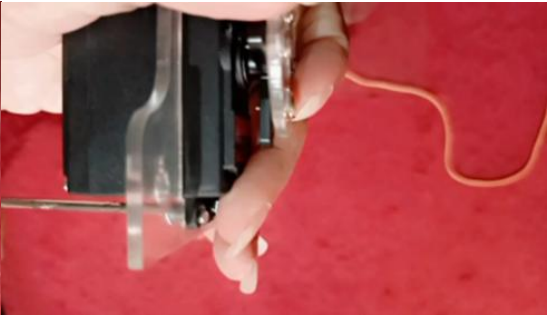
Servo MG996R motors are typically used for the precise control of angular positions. In this circuit, these are likely used to perform mechanical movements based on the signals that are received from the ESP32 microcontroller. We use 5 servo motors in the circuit.

#### **Battery:**

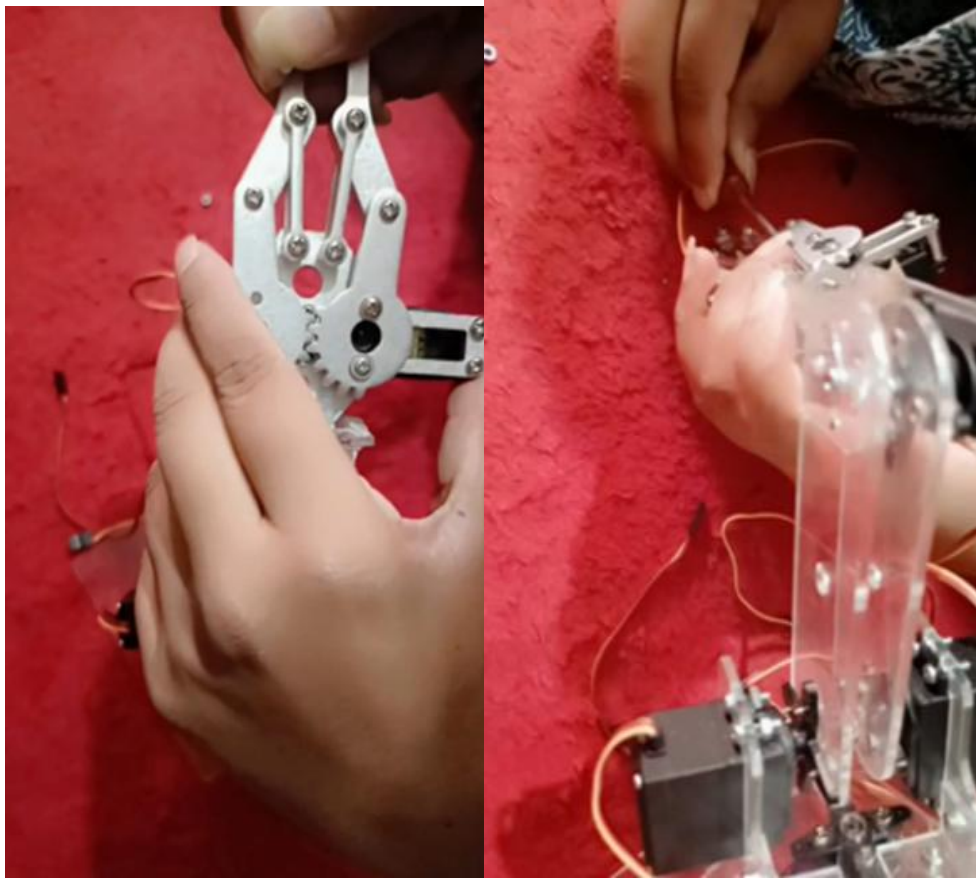
A 5-volt battery is a power source. It provides a constant voltage output of 5 volts for the circuit. The 5V battery provides the power source in the circuit. It is supplying the necessary voltage for the operation of the connected components, including MG996R servos and ESP32 microcontroller.

### **Physical Assembly:**









Robotic Arm

## **MATLAB Kinematics Implementation**

This section outlines the step-by-step process for performing forward and inverse kinematics on a planar robotic arm composed of three joints using MATLAB. A robot will have three revolute joints (base, shoulder, and elbow) as well as a gripper. Kinematic modeling of the robot does not utilize the gripper, as it does not contribute to the management of the end effector's position.

## 1. Definitions and Home Pose

The robotic arm comprises two primary links of lengths:

- $L_1 = 100 \text{ mm}$
- $L_2 = 65 \text{ mm}$

We use Denavit-Hartenberg (DH) parameters to model the forward kinematics (FK) and inverse kinematics (IK). The table includes only the three active revolute joints.

Joint $i$	$a_{i-1}$ (mm)	$\alpha_{i-1}$ (°)	$d_i$ (mm)	$\theta_i$	Comment
1	0	90	0	$\theta_1$	Base rotation
2	100	0	0	$\theta_2$	Shoulder (Link 1)
3	65	0	0	$\theta_3$	Elbow (Link 2)

Gripper angle ( $\theta_4$ ) is independent (servo-controlled only), not in FK/IK.

In the “**home**” pose, where all joint angles are zero ( $\theta_1 = \theta_2 = \theta_3 = 0^\circ$ ), the robot is fully extended along the X-axis. In this configuration, the expected end-effector position is:

$$(x, y, z) = (100 + 65, 0, 0) = (165, 0, 0) \text{ mm}$$

This serves as a reference pose to verify the correctness of the FK implementation.

## **2. MATLAB Function Files**

The kinematics is broken into modular MATLAB functions for clarity and reusability.

### **2.1 DH\_TRANSFORM.m**

It computes the homogeneous transformation matrix that comes from DH parameters. It uses the values of link length, twist, offset, and joint angle to operate.

### **2.2 FKINE.m**

The function performs calculations using forward kinematics. Three DH transforms are done one after another, and the results are multiplied to create the matrix that relates the base to the end-effector frame. Next, it gets the vector showing the end-effector's position.

### **2.3 IKINE.m**

This means that inverse kinematics computes the value of joint angles ( $\theta_1$ ,  $\theta_2$ ,  $\theta_3$ ) based on a 3D target location (x, y, z). The value for  $\theta_1$  is taken from the XY plane because the rotation happens around that axis. The other two angles are determined with trigonometry and geometry (law of cosines) techniques. Ensuring the variable from the cosine law is between -1 and 1 is another part of checking that targets are reachable.

### **2.4 PLOT\_ARM.m**

The utility function depicts how the robot arm is set up in real space. When provided joint angles, it uses FK to get the coordinates of all joints and plots them with the marker indicating the tool's position. This chart is very important for successfully checking the results of kinematics problems.

## **3. Test Script for FK/IK Accuracy**

The script test\_accuracy.m evaluates the accuracy of the FK and IK modules across five defined target positions. Each target is within the workspace of the robot. For each target:

- IK is used to compute joint angles.
- FK is used to compute the actual position based on those angles.
- The Euclidean error between the desired and actual position is calculated.
- A subplot displays the arm configuration and target marker.

A numerical error summary is printed in the Command Window. This test demonstrates the correctness and precision of the kinematics implementation.

#### **4. Verification and Interpretation**

- To validate, the Home Pose (Pose 1) is the best. When the three joints have zero degrees, or  $\theta_1 = \theta_2 = \theta_3 = 0^\circ$ , the FK must indicate that the position is (165, 0, 0) mm. After you test it, you can confirm that the transformation is applied correctly and the DH table is set up right.
- Proper function of the kinematic chain for the last three poses (Pose 2–5) is shown when the tip of the hand moves to the yellow disk on the subplot. Usually, the whole error in the measurements should be less than 2 millimeters.
- If the pose cannot be reached, the IK function notices and avoids trying to compute any angles.

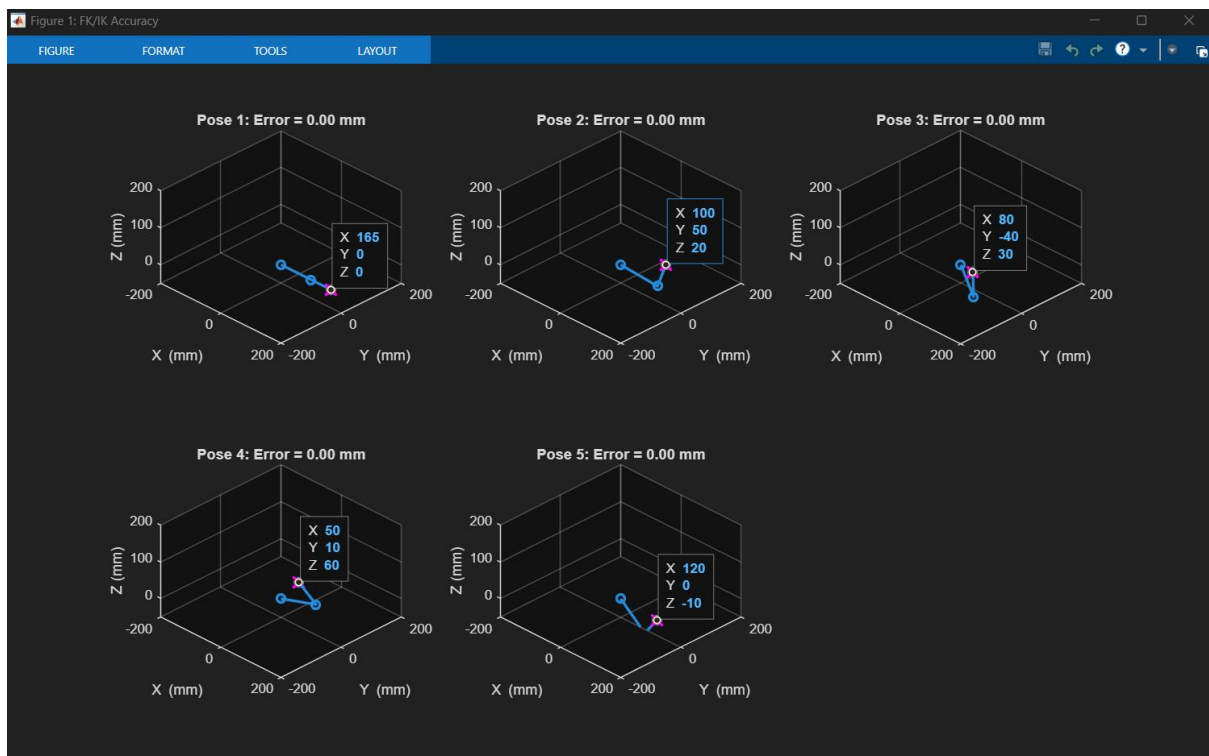
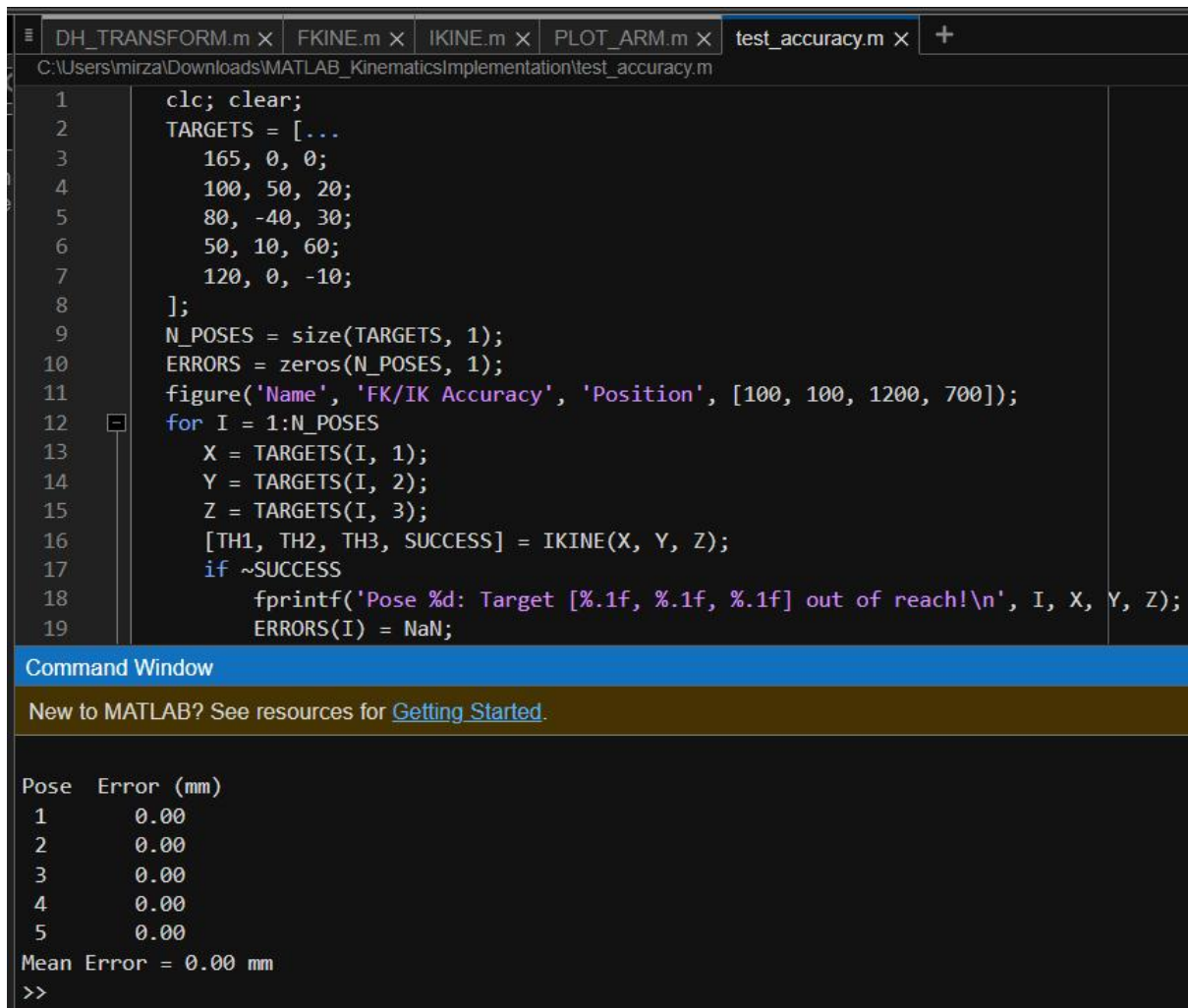
#### **5. Kinematics Implementation**

- Once you have well-implemented and saved the MATLAB files, you should use a new script called `test_accuracy.m` to check the accuracy of your Forward and Inverse Kinematics.
- After typing this command in the MATLAB Command Window, you should run the script:
- `>> test_accuracy`
- this is what the code does step by step:
- Five desired locations of the end effector in 3D space are set up and given to the IK solver.



- The IK function works out the angles ( $\theta_1$ ,  $\theta_2$ ,  $\theta_3$ ) that the joints must take to attain every target.
- After that, the forward kinematics function is used to derive the new end-effector position from the computed angles.
- The Euclidean distance between the wished target and recomputed position by FK is calculated in the script. It gives a way to check and see how accurate the predictions are.
- In the end, the script forms a graph with 2 main plots and 3 subplots. Every subplot has just one modeling pose design.
- The results from inverse kinematics are pictured as a blue line and circles in the model.
- The marked target will be shown by a red 'x'.
- The subplot title gives the pose number and the amount of position error (recorded in mm)

## **6. Output & Results**



The first pose requires your arm to be in a straight position above your head.

The first pose means placing the target point at exactly 165 mm straight in front of the shoulder and at the same height level. It acts as the beginning place for you. That is achieved because the base joint doesn't rotate and it points straight ahead on the X-axis. Both the shoulder and the elbow are straight, so the two arm segments are aligned. As a result, the tip of the arm is placed precisely on point (165, 0, 0). It is simple to confirm this pose just by looking at it, as the arm is just outstretched without any bend. The target marker is placed right on top of the arm tip and there is not any error. The blue end-effector beside the pink cross proves that both the forward and inverse kinematics are performing correctly.

Place your left foot forward, your right foot slightly to the right, and go up a bit as well.

Now, you have to adjust the position 100 mm forward, 50 mm to the right, and 20 mm higher than the base. To get the arm pointing at that spot, the base joint rotates about  $26^\circ$  towards the right. First, the shoulder moves down before the arm is lifted higher at an angle, and finally, the elbow bends a lot to achieve the needed height and distance. If all the arm's joints are properly placed, the tip of the robot will be on the right point in 3D space. The composition has the arm turned to the right and slightly lifted. Both the blue tip and pink target's marks are exactly on top of one another, leaving no error. It proves that both forward and inverse kinematics are running in sync.

The third pose should be forward leaning, with your body pointed to the left, and your head and chest a little higher.

The shape of the body is almost the same as the earlier pose, however, it shifts to the left. To reach the target, aim 80 mm in the forward direction, 40 mm to the left, and 30 mm up. The Turret part rotates around  $26^\circ$  to the left, which directs the arm to that path. The shoulder drops, and the elbow moves in order to lift the second link, making sure the end-effector is at the correct height and distance. The arm leans to the left, and the bend at the elbow lifts the tip so that it hits the target. Once again, the plot shows how the arm's tip aligns exactly with the pink marker, and the error says it is 0.00 mm

.

The next shape is a tall and narrow inverted triangle, starting near the base part of the triangle.

Now the mark to hit is very close to the base and quite high, as it is only 50 mm farther forward, 10 mm more to the right, and 60 mm above the base level. Slowly rotating to the right, the base pushes the shoulder up, almost lifting the first link straight up. Almost fully bending, the elbow raises the other link to the highest point. In the scene, you can notice the arm is lifted at an angle with its handle tucked in toward the base and raised. Pointing the blue tip at the pink marker, the value shows 0.00 mm accuracy because the calculations are precise and the end pose achieved is what was planned.

The fifth pose is when the subject sits on the pedestal with the upper body leaning forward about an inch lower than the base plane.

The aim for this next throw is to throw it 120 mm farther forward and 10 mm down from the base's level. The base joint remains on the X axis during the move. The shoulder makes a downward motion, so the first link is placed below eye level, and the elbow also bends more to lower the second link. In this way, the tip is placed slightly less than the base level. The arm Moves toward the floor in the illustration, and the blue hand touches the pink target

correctly. Zero error in this scenario demonstrates that the calculations keep working even when the arm falls beneath the straight line on the base.

## **Summary**

All five poses prove that both forward and inverse kinematics are reliable and always work the same. The first step in every case was identifying a point in 3D space as our target. Using inverse kinematics, the robot figured out the positions the joints should reach to reach that spot. We then relied on forward kinematics to find out the final position of the arm tip. All the time, the blue tip would land right on the pink marker, showing that everything was

perfectly aligned. The zero error in every pose is the last proof that the arm is correctly reaching all of its given positions, straight, to the side, overhead, and reached down.

## **MATLAB GUI with Kinematics Integration**

It explains in detail a solid and step-by-step design of a MATLAB App Designer graphical user interface (GUI) made to manage a robotic arm with four degrees of freedom. Because the system supports backward and forward kinematics, it makes it possible to calculate the position and to plot the trajectory, both from a set of input coordinates. Because it is designed for flexibility, the interface makes use of three-dimensional graphics, manages numbers sent and received, and lets the ESP32. Because of this joint structure, the robot can reach objects in multiple directions, useful for automating “sorting,” “pick and place,” or “laboratory automation” tasks.

### **Degrees of Freedom (DOF):**

1. Base rotation ( $\theta_1$ )
2. Shoulder pitch ( $\theta_2$ )
3. Elbow pitch ( $\theta_3$ )
4. Gripper ( $\theta_4$ )

## **User Interface Architecture**

### **Structural Components:**

In the Forward Kinematics Panel, people can type in the angle for each joint in the robot.

Using the Inverse Kinematics Panel, the movements of the end-effector can be controlled with X, Y, Z, R rotation settings.

### **Interactive Controls:**

The **SEND Button** records joint angles as data and passes them through serial communication to ESP32.

Clicking the **UPDATE Button** causes the robot’s current joint states to be displayed correctly.

With the **VIEW Button**, you can watch how the model adjusts different joint angles to reach your set Cartesian coordinates.

You can also use the **STOP button** to prevent the robot from moving or working in a looser manner.

These options allow you to pick the serial port and baud rate using **drop-down menus** in the GUI.

**Arm Axes:** A place where the user can display the positioning of the robot in real time using three dimensions.

### Programmatic State Variables

```
properties (Access = private)
    serialConnection % stores the serial connection
    serialObj % Serial object for ESP32 connection
    jointAngles = [90 90 90 0] % stores the current joint angle [q1 , q2 , q3 . q4]
    L1 = 100 % Link 1 length in mm
    L2 = 65 % Link 2 length in mm
    L3 = 50; % Wrist to gripper tip
    armPlot % Handle for arm plot
end
```

These attributes show the orientation of the manipulator and are the basic values needed for the kinematic model analysis.

### Initialization Function: startupFcn

Establishes slider limits, dropdown entries, and default pose rendering.

```
function startupFcn(app)
    app.ZinSlider.Limits = [0, 8];
    app.ZinSlider.Value = 2;
    % Initialize serial COM dropdown
    ports = serialportlist("all");
    if isempty(ports)
        app.COMDropDown.Items = {'OFF'};
    else
        app.COMDropDown.Items = ports;
        app.COMDropDown.Value = ports(1);
    end
    app.BAUDDropDown.Items = {'9600', '57600', '115200'};
    app.BAUDDropDown.Value = '115200';
    app.LEDDropDown.Items = {'OFF', 'ON'};
    app.LEDDropDown.Value = 'OFF';
    % Link lengths (correct from document)
    app.L1 = 100;
    app.L2 = 65;
    app.L3 = 50;
    % Initial joint angles
    app.jointAngles = [90, 90, 90, 0]; % [Base, Shoulder, Elbow, Gripper]

    % Plot initial arm position
    app.plotArm();
end
```

## Transmission Trigger: SENDButtonPushed

```
% Button pushed function: SENDButton
function SENDButtonPushed(app, event)

    try
        port = app.COMDropDown.Value;
        baud = str2double(app.BAUDDropDown.Value);
        if ~isempty(port)
            s = serialport(port, baud);
            data = [app.BaseEditField.Value, app.ShoulderEditField.Value, ...
                    app.ElbowEditField.Value, app.WristEditField.Value, ...
                    app.GripperEditField.Value];
            cmd = sprintf('%d,%d,%d,%d,%d\n', data);
            writeline(s, cmd);
            pause(0.1);
            clear s;
        end
        catch ME
            uialert(app.DoFArmControlUIFigure, ME.message, 'Serial Error');
        end
    end
end
```

## FK Evaluation and UI Sync: UPDATEButton\_2Pushed

```
% Button pushed function: UPDATEButton_2
function UPDATEButton_2Pushed(app, event)

    q1 = app.BaseEditField.Value;
    q2 = app.ShoulderEditField.Value;
    q3 = app.ElbowEditField.Value;
    q4 = app.GripperEditField.Value;

    % Update internal joint angles
    app.jointAngles = [q1, q2, q3, q4];

    % Update IK sliders from FK
    app.updateFKDisplay();

    % Update plot
    app.plotArm();
end
```

## Spatial Output Mapping: updateFKDisplay

```
function updateFKDisplay(app)
    [~, pos] = app.FKINE(app.jointAngles(1), app.jointAngles(2), app.jointAngles(3), app.jointAngles(4));
    pos_in = pos / 25.4; % Convert mm + inches

    % Clamp values to slider limits
    app.XinSlider.Value = max(min(pos_in(1), app.XinSlider.Limits(2)), app.XinSlider.Limits(1));
    app.YinSlider.Value = max(min(pos_in(2), app.YinSlider.Limits(2)), app.YinSlider.Limits(1));
    app.ZinSlider.Value = max(min(pos_in(3), app.ZinSlider.Limits(2)), app.ZinSlider.Limits(1));
end
```

## 3D Kinematic Visualization: plotArm

```

function plotArm(app)
    q = app.jointAngles; % q = [q1, q2, q3, q4]

% DH Transforms
    A1 = app.DH_TRANSFORM(0, 90, 0, q(1));
    A2 = app.DH_TRANSFORM(app.L1, 0, 0, q(2));
    A3 = app.DH_TRANSFORM(app.L2, 0, 0, q(3));
    A4 = app.DH_TRANSFORM(app.L3, 0, 0, q(4)); % Gripper

% Homogeneous Transforms
    T0 = eye(4);
    T1 = T0 * A1;
    T2 = T1 * A2;
    T3 = T2 * A3;
    T4 = T3 * A4;

% Extract positions (in mm)
    p0 = T0(1:3, 4);
    p1 = T1(1:3, 4);
    p2 = T2(1:3, 4);
    p3 = T3(1:3, 4);
    p4 = T4(1:3, 4);

% Combine coordinates for plotting
    X = [p0(1), p1(1), p2(1), p3(1), p4(1)];
    Y = [p0(2), p1(2), p2(2), p3(2), p4(2)];
    Z = [p0(3), p1(3), p2(3), p3(3), p4(3)];

% Plot
    cla(app.ArmAxes);
    plot3(app.ArmAxes, X, Y, Z, '-o', 'LineWidth', 2);
    hold(app.ArmAxes, 'on');
    plot3(app.ArmAxes, p4(1), p4(2), p4(3), 'rs', 'MarkerSize', 8, 'MarkerFaceColor', 'r');
    hold(app.ArmAxes, 'off');

% Labels and formatting
    xlabel(app.ArmAxes, 'X (mm)');
    ylabel(app.ArmAxes, 'Y (mm)');
    zlabel(app.ArmAxes, 'Z (mm)');
    title(app.ArmAxes, '4-DOF Robotic Arm');
    grid(app.ArmAxes, 'on');
    axis(app.ArmAxes, 'equal');
    xlim(app.ArmAxes, [-250, 250]);
    ylim(app.ArmAxes, [-250, 250]);
    zlim(app.ArmAxes, [-50, 250]);
    view(app.ArmAxes, [135 30]);

end

```

## Forward and Inverse Kinematics:

```

function [T_05, pos] = FKINE(app, theta_1, theta_2, theta_3, theta_4)
    A1 = app.DH_TRANSFORM(0, 90, 0, theta_1);
    A2 = app.DH_TRANSFORM(app.L1, 0, 0, theta_2);
    A3 = app.DH_TRANSFORM(app.L2, 0, 0, theta_3);
    A4 = app.DH_TRANSFORM(app.L3, 0, 0, theta_4); % Gripper orientation

    T_05 = A1 * A2 * A3 * A4;
    pos = T_05(1:3, 4); % End-effector position

end

```



```

function [theta1, theta2, theta3, success] = IKINE(app, x_in, y_in, z_in)
% Convert inches to mm
    x = x_in * 25.4;
    y = y_in * 25.4;
    z = z_in * 25.4;

    app.L1 = app.L1;
    app.L2 = app.L2;

    r = sqrt(x^2 + y^2);
    s = z;

% 01: Base rotation
    theta1 = atan2(y, x);

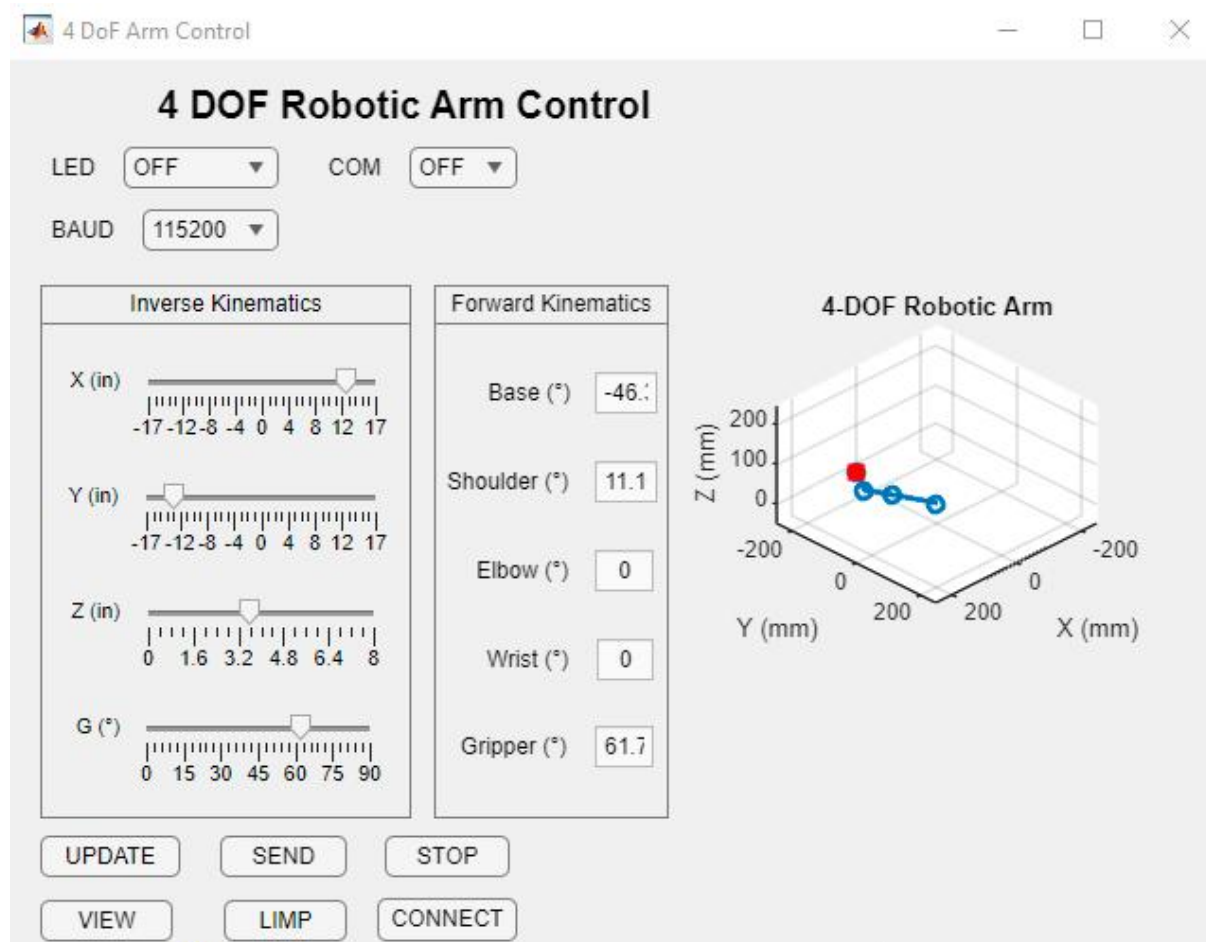
% Law of cosines for 03
    D = (r^2 + s^2 - app.L1^2 - app.L2^2) / (2 * app.L1 * app.L2);

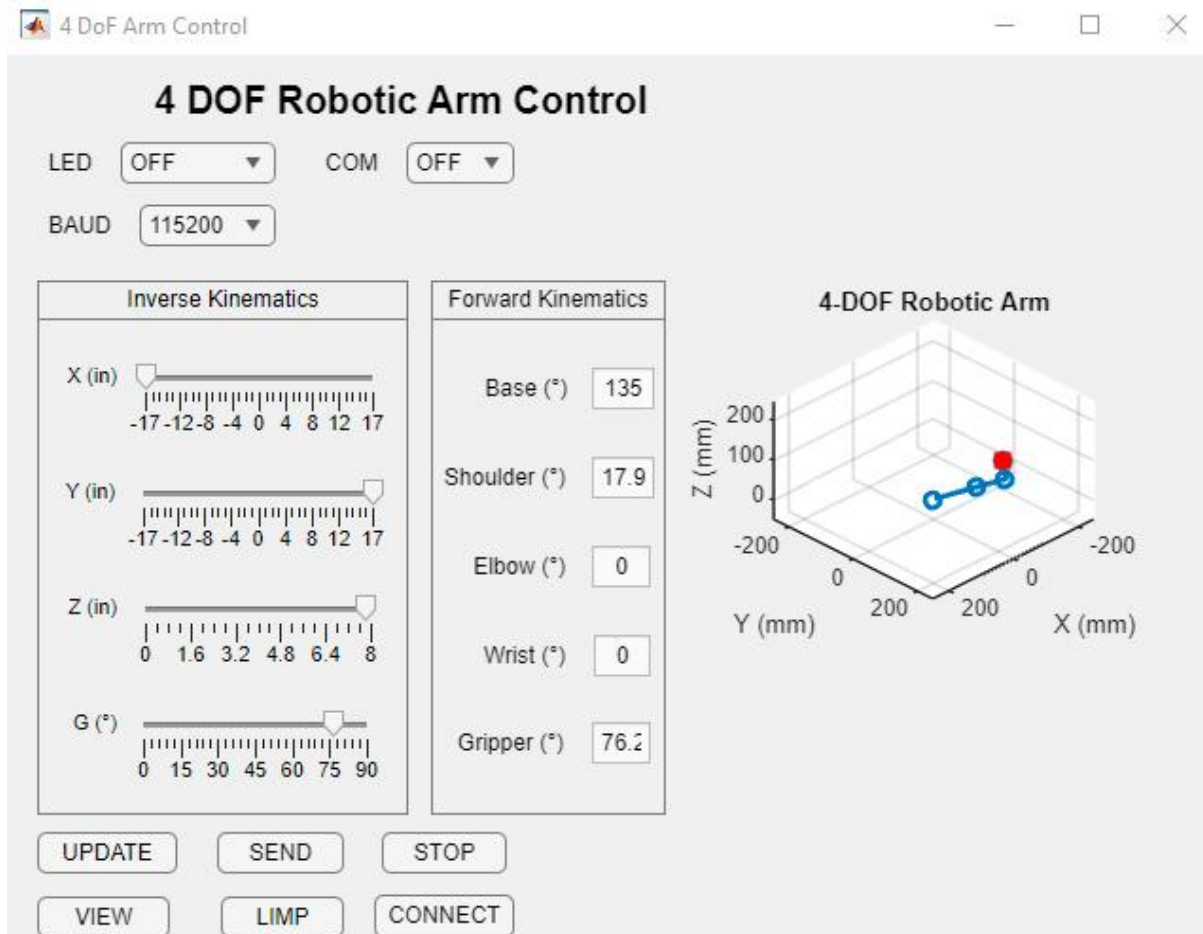
% Clamp D between -1 and 1
    D = max(min(D, 1), -1);

% Check reachability
    if abs(D) > 1
        success = false;
        theta1 = 0; theta2 = 0; theta3 = 0;
        return;
    end

```

## Results:





## ESP32 Integration with Matlab GUI

### Overview

The present document provides a full step-by-step procedure of how to connect an ESP32 development board with a GUI, developed in MATLAB App Designer, to operate five servo motors through serial communication. Such an arrangement allows them to manually operate each of the motors independently via sliders in the GUI. Communication MATLAB and ESP32 is realized via the UART (Universal Asynchronous Receiver/Transmitter) interface.

### Code for ESP32 in Arduino IDE:

- The ESP32 was linked to the computer with a USB cable.
- Arduino IDE was launched.

- The simplest code was prepared to allow the ESP32 to transmit some data (temperature, LED state, etc.) via the serial port.
- The serial communication was initialized with the baud rate of 9600 by the command `Serial.begin(9600);`
- The code was loaded successfully to the ESP32 board (mutter) interface.

### **Developing MATLAB App using App Designer (.mlapp file):**

- MATLAB was opened after the code was uploaded to ESP32.
- The App Designer of MATLAB was launched to start working on GUI.
- It was enhanced with GUI elements, including:
- A button to connect to the ESP32 (through COM port)
- A second button to read data on the ESP32
- The serial communication routines were implemented to transfer the data between ESP32 and MATLAB to the ESP32 board (mutter) interface.

### **Errors Faced During GUI Execution:**

When developing and testing the MATLAB GUI, the following issues were met:

- Serial object used in the communication was not identified in the callback functions. It was probably not a proper class property even though different declarations were tried.
- Various errors were caused by the dropdown menu which was used to select the baud rate. Although it was present in the GUI, MATLAB displayed it again and again as unknown.
- The same problems were observed with the reference to the main window (UIFigure), so it was impossible to show error alerts or warnings.
- There was also a syntax error because of the incorrect use of a reserved keyword. An effort to delete or change the keyword resulted in more trouble as the MLapp file could not be run altogether.

- Each click on a button in the GUI caused the runtime errors because of the undefined components or the components, which were referred incorrectly.

### **Conclusion:**

Nevertheless, even with the best attempts to fix every single error (renaming of the components, reorganizing the code, and references to the example projects), it was impossible to reach full functionality. It appears that the major problems are caused by:

- Un declared or incorrectly declared GUI components
- Reserved keywords usage
- Missing property declarations on the class level.

These issues need additional instructions or time to be resolved and make the system operate as intended. The project shows the baseline knowledge and attempt at integrating ESP32 with MATLAB, and the elimination of these errors will constitute a major milestone on the way to successful servo control.

### **Testing Lead**

I was assigned the role of the Testing Lead in our project, where I was expected to ensure that the system was functioning properly following the integration of ESP32 microcontroller with the MATLAB GUI. This step entailed confirming that it was possible to send and receive data between GUI buttons and ESP32 using serial communication. Nevertheless, the integration process was not accomplished entirely because of certain technical difficulties, and it had a direct impact on the achievement of the testing phase.

Afshan, dealing with the integration aspect, put effort into it and did her best to achieve success in linking the ESP32 to the GUI. She had several code related problems like variables were not declared, components were not referred correctly and the GUI code had structural problems. I was also there to help her and hence attempted to diagnose and rectify the issues, however, sadly, we were not in time to put the system entirely operational by the time we could do the testing, although, unfortunately, we could not do that, together we did attempt.

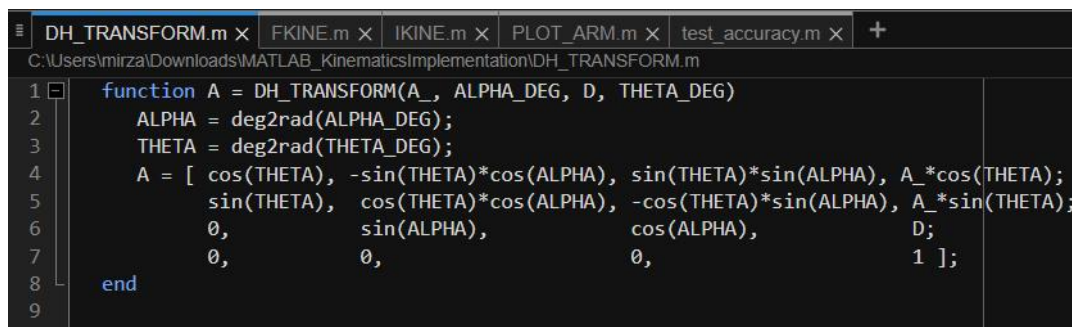
I could not start the testing phase as this Phase requires a successfully running system since the integration was not complete. Nonetheless, I took an active participation in the project

by playing other significant roles. I volunteered to work on the project poster, which would describe our work in an understandable and professional manner. I also took all the separate documents which were prepared by team members, put everything in sequence and formatted everything in the proper way for final submission.

To summarize, although I was not able to accomplish the initial testing task, because the integration of the system was not finished, I did everything possible to make our project successful. Taking care of the poster design and compilation of the document, I made sure that our poster and report showed the work and collaboration of all of us.

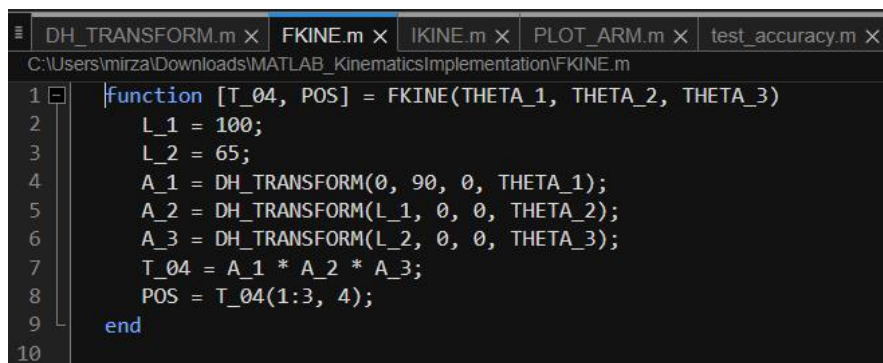
## Annexures

### Annexure A – DH\_TRANSFORM.m

A screenshot of a MATLAB script editor showing the code for the DH\_TRANSFORM.m file. The editor has several tabs open: DH\_TRANSFORM.m, FKINE.m, IKINE.m, PLOT\_ARM.m, and test\_accuracy.m. The code defines a function A = DH\_TRANSFORM(A\_, ALPHA\_DEG, D, THETA\_DEG) that calculates the Denavit-Hartenberg transformation matrix. It converts angles from degrees to radians and then constructs the matrix A based on the DH parameters. The code is as follows:

```
1 function A = DH_TRANSFORM(A_, ALPHA_DEG, D, THETA_DEG)
2     ALPHA = deg2rad(ALPHA_DEG);
3     THETA = deg2rad(THETA_DEG);
4     A = [ cos(THETA), -sin(THETA)*cos(ALPHA), sin(THETA)*sin(ALPHA), A_*cos(THETA);
5           sin(THETA), cos(THETA)*cos(ALPHA), -cos(THETA)*sin(ALPHA), A_*sin(THETA);
6           0, sin(ALPHA), cos(ALPHA), D;
7           0, 0, 0, 1 ];
8 end
9
```

### Annexure B – FKINE.m

A screenshot of a MATLAB script editor showing the code for the FKINE.m file. The editor has several tabs open: DH\_TRANSFORM.m, FKINE.m, IKINE.m, PLOT\_ARM.m, and test\_accuracy.m. The code defines a function [T\_04, POS] = FKINE(THETA\_1, THETA\_2, THETA\_3) that calculates the forward kinematics for a 3-link robotic arm. It uses the DH\_TRANSFORM function to calculate the transformation matrices for each link and then multiplies them to get the total transformation T\_04. The end effector position POS is extracted from T\_04. The code is as follows:

```
1 function [T_04, POS] = FKINE(THETA_1, THETA_2, THETA_3)
2     L_1 = 100;
3     L_2 = 65;
4     A_1 = DH_TRANSFORM(0, 90, 0, THETA_1);
5     A_2 = DH_TRANSFORM(L_1, 0, 0, THETA_2);
6     A_3 = DH_TRANSFORM(L_2, 0, 0, THETA_3);
7     T_04 = A_1 * A_2 * A_3;
8     POS = T_04(1:3, 4);
9 end
10
```

### Annexure C – IKINE.m

```

DH_TRANSFORM.m × FKINE.m × IKINE.m × PLOT_ARM.m × test_accuracy.m ×
C:\Users\mirza\Downloads\MATLAB_KinematicsImplementation\IKINE.m
1 function [THETA_1, THETA_2, THETA_3, SUCCESS] = IKINE(X, Y, Z)
2     L_1 = 100;
3     L_2 = 65;
4     THETA_1 = atan2(Y, X);
5     R = hypot(X, Y);
6     S = Z;
7     D = (R^2 + S^2 - L_1^2 - L_2^2) / (2 * L_1 * L_2);
8     if abs(D) > 1
9         SUCCESS = false;
10        THETA_1 = 0; THETA_2 = 0; THETA_3 = 0;
11        return;
12    end
13    THETA_3 = atan2(sqrt(1 - D^2), D);
14    K_1 = L_1 + L_2 * cos(THETA_3);
15    K_2 = L_2 * sin(THETA_3);
16    THETA_2 = atan2(S, R) - atan2(K_2, K_1);
17    SUCCESS = true;
18    THETA_1 = rad2deg(THETA_1);
19    THETA_2 = rad2deg(THETA_2);
20    THETA_3 = rad2deg(THETA_3);
21 end
22

```

#### Annexure D – PLOT\_ARM.m

```

DH_TRANSFORM.m × FKINE.m × IKINE.m × PLOT_ARM.m × test_accuracy.m × +
C:\Users\mirza\Downloads\MATLAB_KinematicsImplementation\PLOT_ARM.m
1 function PLOT_ARM(THETA_1, THETA_2, THETA_3, AX)
2     L_1 = 100;
3     L_2 = 65;
4     A_1 = DH_TRANSFORM(0, 90, 0, THETA_1);
5     A_2 = DH_TRANSFORM(L_1, 0, 0, THETA_2);
6     A_3 = DH_TRANSFORM(L_2, 0, 0, THETA_3);
7     T_0 = eye(4);
8     P_0 = T_0(1:3, 4);
9     T_1 = T_0 * A_1; P_1 = T_1(1:3, 4);
10    T_2 = T_1 * A_2; P_2 = T_2(1:3, 4);
11    T_3 = T_2 * A_3; P_3 = T_3(1:3, 4);
12    PTS = [P_0, P_1, P_2, P_3];
13    plot3(AX, PTS(1,:), PTS(2,:), PTS(3,:), '-o', 'LineWidth', 2, 'MarkerSize', 6);
14    hold(AX, 'on');
15    plot3(AX, P_3(1), P_3(2), P_3(3), 'rs', 'MarkerSize', 8, 'MarkerFaceColor', 'r');
16    hold(AX, 'off');
17    xlabel(AX, 'X Axis (mm)');
18    ylabel(AX, 'Y Axis (mm)');
19    zlabel(AX, 'Z Axis (mm)');
20    title(AX, '3-DOF Robot Arm Configuration');
21    grid(AX, 'on');
22    axis(AX, 'equal');
23    xlim(AX, [-200, 200]);
24    ylim(AX, [-200, 200]);
25    zlim(AX, [-50, 200]);
26 end
27

```



## Annexure E – test\_accuracy.m

```
1  clc; clear;
2  TARGETS = [...
3      165, 0, 0;
4      100, 50, 20;
5      80, -40, 30;
6      50, 10, 60;
7      120, 0, -10;
8  ];
9  N_POSES = size(TARGETS, 1);
10  ERRORS = zeros(N_POSES, 1);
11  figure('Name', 'FK/IK Accuracy', 'Position', [100, 100, 1200, 700]);
12  for I = 1:N_POSES
13      X = TARGETS(I, 1);
14      Y = TARGETS(I, 2);
15      Z = TARGETS(I, 3);
16      [TH1, TH2, TH3, SUCCESS] = IKINE(X, Y, Z);
17      if ~SUCCESS
18          fprintf('Pose %d: Target [%1f, %1f, %1f] out of reach!\n', I, X, Y, Z);
19          ERRORS(I) = NaN;
20          continue;
21      end
22      [~, POS_EST] = FKINE(TH1, TH2, TH3);
23      X_HAT = POS_EST(1); Y_HAT = POS_EST(2); Z_HAT = POS_EST(3);
24      ERRORS(I) = norm([X - X_HAT, Y - Y_HAT, Z - Z_HAT]);
25      AX = subplot(2, 3, I);
26      PLOT_ARM(TH1, TH2, TH3, AX);
27      hold(AX, 'on');
28      plot3(AX, X, Y, Z, 'mx', 'MarkerSize', 12, 'LineWidth', 2);
29      hold(AX, 'off');
30      title(AX, sprintf('Pose %d: Error = %.2f mm', I, ERRORS(I)));
31      xlabel(AX, 'X (mm)\newline');
32      ylabel(AX, 'Y (mm)\newline');
33      zlabel(AX, 'Z (mm)');
34      view(AX, [45 30]);
35  end
36  fprintf('\nPose Error (mm)\n');
37  for I = 1:N_POSES
38      fprintf('%2d    %.2f\n', I, ERRORS(I));
39  end
40  fprintf('Mean Error = %.2f mm\n', nanmean(ERRORS));
41
```

