SRI VENKATESWARA INTERNSHIP PROGRAM FOR RESEARCH IN ACADEMICS (SRI-VIPRA)

Student Internship

SRI-VIPRA

Project Report of 2025: SVP-2516

Smart-Sense Living: An Adaptive IoT Control Prototype

IQAC

Sri Venkateswara College

University of Delhi

Benito Juarez Road, Dhaula Kuan, New Delhi

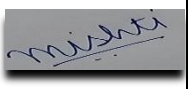
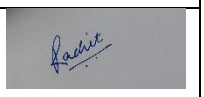
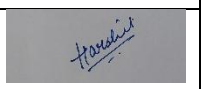
New Delhi -110021

SRIVIPRA PROJECT 2025

Title: Smart-Sense Living: An Adaptive IoT Control Prototype

Name of Mentor: Dr Hari Singh **Name of Department: Electronics** **Designation: Assistant Professor**	**Photo** 

List of students under the SRIVIPRA Project

S.No	Photo	Name of the student	Roll number	Course	Signature
1		Mishti Angirash	1624003	B.Sc. Hons. Electronics Semester II	
2		Rachit Singh	1624006	B.Sc. Hons. Electronics Semester II	
3		Harshit Gupta	1624053	B.Sc. Hons. Electronics Semester II	

Signature of Mentor

CERTIFICATE OF ORIGINALITY

This is to certify that the aforementioned students from Sri Venkateswara College have participated in the summer project **SVP- 2516** titled **Smart-Sense Living: An Adaptive IoT Control Prototype.** The participants have carried out the research project work under my guidance and supervision **from 1st July 2025 to 30th September 2025.** The work carried out is original and carried out in an online/offline/hybrid mode.

Signature of Mentor

ACKNOWLEDGEMENT

First and foremost, thanks to the ALMIGHTY for giving us the strength, patience, and knowledge that enabled us to complete the project.

We convey our deepest gratitude to our Principal, Prof. Vajala Ravi, for giving us such an opportunity to carry out the research project work under the Sri VIPRA Internship Program 2025. We would also like to extend our sincere thanks to Prof. Varthika Mathur (Coordinator, IQAC), and SRIVIPRA Coordinators Dr. P. Jayaraj (Sciences) and Dr. Haokam Vaiphe (Humanities and Commerce) for their continuous support throughout the project.

We are grateful to the Department of Electronics for encouraging us to widen our academic perspectives. We owe a deep sense of gratitude to our mentor, Dr Hari Singh, for his constant support and guidance. It was his encouragement and assistance that made this work possible. During these weeks, we learnt about the functioning of gesture control systems, smart devices, and their real-time interaction.

Lastly, we would like to thank all those people who have been associated with this project and have helped us with it. Immeasurable appreciation and deepest gratitude to all of you for your help and support.

TABLE OF CONTENTS

Chapter 1-Introduction

Throughout history, humanity's greatest pursuit has been to make life easier, more comfortable, and more efficient. From the invention of the wheel to the rise of the internet, every major innovation has been driven by the same desire—to simplify everyday life and improve the human experience. As societies evolve, so do our expectations. The modern world moves at an incredible pace, and people increasingly seek solutions that save time, reduce effort, and seamlessly blend into their daily routines. This constant need for convenience has led to the creation of luxurious products and transformative technologies that redefine how we live, work, and interact with our surroundings. In the 21st century, technology has become an inseparable part of our lives. It is no longer limited to industrial applications or computing laboratories but has entered the very fabric of our homes. The machines we once operated manually have now become intelligent companions that anticipate our needs and respond to our commands. From smartphones that manage our schedules to smart refrigerators that track our groceries, technology has redefined domestic life. The desire for speed and efficiency, combined with an unending curiosity to innovate, has allowed us to develop systems that can perform complex tasks almost instantaneously. As a result, humans now have the freedom to think creatively, explore new possibilities, and focus on innovation that enhances everyday living.

One of the most fascinating outcomes of this technological evolution is the concept of the smart home. The idea of automating household tasks was once a futuristic dream, confined to science fiction. Today, it is a reality made possible by the integration of sensors, wireless communication, and intelligent control systems. Smart home technology has not only simplified the way we live but also reshaped our relationship with our environment. We can now control lights, fans, air conditioners, and security systems with the tap of a button—or even without touching anything at all. However, this growing convenience also raises new expectations. People no longer want to adapt themselves to technology; they want technology to adapt to them [1].

In this pursuit of seamless interaction between humans and machines, we have developed systems that respond to the most natural forms of human communication—voice, facial expressions, and gestures. These modes of interaction bridge the gap between humans and devices, making technology feel more intuitive and human-centred. The move away from traditional switches and remotes toward more instinctive forms of control reflects a major shift in technological design philosophy. It signifies a step toward creating environments that understand us, rather than ones

that merely obey commands. Further, voice-controlled systems have already revolutionised the way we interact with technology in our homes. Devices like smart speakers and digital assistants have made it possible to manage everything—from turning on the lights to playing music—simply by speaking. However, as revolutionary as these systems are, they are not without limitations. Voice recognition depends heavily on clarity of speech and a quiet environment. In noisy surroundings, or for individuals with speech impairments, such systems may not function effectively. These challenges highlight the need for an alternative mode of interaction that is both reliable and inclusive.

This is where gesture-based control enters the picture. Gestures are among the most natural forms of human expression. A simple wave, nod, or motion of the hand can communicate an entire message without uttering a single word. Integrating gesture recognition into smart home systems introduces a new level of accessibility and intuitiveness. It eliminates the dependency on physical contact or verbal instructions, offering users the freedom to interact effortlessly with their environment [2]. Our project, **Smart-Sense Living**, is rooted in this very idea. It is designed to empower users with a touchless, gesture-controlled system that makes home automation more convenient, inclusive, and human-centred. With Smart-Sense Living, everyday actions—such as turning on a fan, switching off lights, or adjusting room temperature—can be performed with simple hand movements. This not only enhances comfort and hygiene but also ensures usability for people of all ages and abilities, including those who may find voice or touch-based systems challenging to use.

Beyond convenience, Smart-Sense Living represents a broader vision that is to create a more personal and intuitive interaction between humans and technology. It transforms the home from a static living space into a responsive ecosystem that reacts to human presence, motion, and needs. Such innovations also have broader social implications—they contribute to sustainable living by optimizing energy use and reducing wastage through intelligent automation.

As we continue to advance technologically, it is crucial to ensure that innovation remains rooted in human experience. Technology should not complicate life but make it simpler, safer, and more meaningful. Gesture-controlled smart systems like Smart-Sense Living are not merely about modernisation but they also represent a step towards a future where technology and humanity coexist harmoniously. They remind us that progress is not just about smarter machines but about creating environments that truly understand and respond to the human touch.

The primary objectives of this project include:

- Developing an intuitive gesture recognition interface that allows seamless control of appliances without the need for physical contact.
- Enhancing accessibility by offering an alternative control method that works effectively in noisy surroundings or for people with speech difficulties.
- Designing a cost-effective and reliable system using commonly available sensors and microcontrollers, making it affordable for widespread use.
- Contributing to the evolution of smart homes by integrating natural human-computer interaction techniques, bringing us closer to truly intelligent living spaces.

Through this project, we aim to create a practical and innovative solution that simplifies everyday life and makes home automation more inclusive, efficient, and future-ready.

Chapter 2-System Design

2.1 Overview of the Software/Programming

The programming part of our project is the most important section because it functions like the brain, controlling everything. Without software, the hardware cannot perform any meaningful task. The software is responsible for detecting hand gestures, recognizing them correctly, and then sending the proper commands to the Arduino, allowing the output to take place.

The entire program has been written in the Python Programming Language. Python was chosen because of its simplicity, readability, and versatility. IT supports powerful libraries that make our work easier. Some of the main libraries we used are *OpenCV, Mediapipe, Kivy, Serial Communication,* and *SocketIO*. These libraries help us design a system that can capture hand gestures in real time, process them quickly, and send the required commands.

The software is designed in such a way that as soon as we open the application, it automatically starts the camera, detects the hand gestures, processes the data, and then sends the correct action. This action can either control system functions like volume or brightness, or be sent to the Arduino board for hardware output. This makes the software the central control system of our project.

Technologies Used

- Python Programming Language – to write the software easily and flexibly.
- OpenCV – for computer vision tasks like detecting hands.
- Mediapipe – for recognizing gestures and tracking hand landmarks.
- Kivy – to create a user-friendly application interface.
- Serial Communication – to send signals to the Arduino.
- SocketIO – for handling communication when needed.

2.2 Overview of the Hardware

The hardware part of our project works closely with the software. If the software is the brain, then the hardware acts like the body which performs the real-world actions. The software detects and processes gestures, while the hardware receives these commands and shows the actual output.

For this project, we used Arduino as the main hardware controller. Arduino was chosen because it is easy to program, readily available, and has the ability to control many types of electronic devices

such as lamps, motors, or other components. It acts as an interface between the computer and the external devices.

Along with Arduino, we also used simple components like resistors and lamps to demonstrate the control of real-world appliances. When a gesture is detected, the command is sent to the Arduino, and the Arduino switches on or off the lamp (or adjusts its brightness) to show how gestures can directly control hardware.

Chapter 3-Software Setup

3.1 Programming Language and Libraries

For this project, we mainly used Python because it is simple, beginner-friendly, and has strong support for computer vision, machine learning, and Graphical User Interface (GUI) development. Python also has a huge community and many libraries, which makes it perfect for developing applications like ours.

Below are the main libraries and tools we used:

- ***Python:*** Python is a high-level, general-purpose programming language created by Guido van Rossum in 1991. It is well known for its easy syntax, readability, and versatility. Python supports multiple programming styles, including procedural, object-oriented, and functional programming. With its large standard library and thousands of third-party packages, it is widely used in web development, artificial intelligence, automation, data science, and more. Being open-source and platform-independent, it is one of the most popular programming languages in the world today .
- ***OpenCV (cv2):*** OpenCV (Open-Source Computer Vision Library) is a powerful open-source library with hundreds of computer vision algorithms [3]. In our project, we used it to capture the live video feed from the camera and process each frame. OpenCV is modular and comes with many useful modules, such as:
    1. Core (core): Basic data structures and functions like Mat arrays.
    2. Image Processing (imgproc): Image filtering, resizing, color conversions, etc.
    3. Image File I/O (imgcodecs): Reading and writing image files.
    4. Video I/O (videoio): Capturing video and handling codecs.
    5. High-Level GUI (highgui): Simple user interface features.
    6. Video Analysis (video): Object tracking, background subtraction, and motion analysis.
    7. Camera Calibration and 3D (calib3d): Stereo vision and 3D reconstruction.
    8. 2D Features Framework (features2d): Feature detection and matching.
    9. Object Detection (objdetect): Pre-trained detectors like faces, eyes, and more.
    10. DNN (dnn): Running deep learning models.
    11. Machine Learning (ml): Classification, regression, clustering.
    12. Photo: Advanced photo editing (denoising, inpainting).

13. Stitching: Panorama creation.

- ***MediaPipe:*** Developed by Google, MediaPipe is an open-source framework that provides ready-to-use AI and ML solutions. In our project, it was used to detect and track hand landmarks (like fingertips and joints) from the video feed. MediaPipe makes it easier to integrate computer vision and machine learning into applications across platforms [4].
- ***Kivy:*** Kivy is an open-source Python library for building multi-platform GUI applications [5]. We used Kivy to design our application's interface so it looks and feels like a proper mobile/desktop app. With Kivy, apps can run on:
    1. Windows, macOS, Linux, and BSD systems.
    2. Android and iOS devices.
    3. Any other touch-enabled devices supporting TUIO.
- ***Serial (PySerial):*** PySerial is a Python library that helps in communication between the computer and external hardware (like Arduino) through USB or COM ports. In our project, it was essential for sending data from the Python program to the Arduino board for controlling outputs.
- ***NumPy:*** NumPy is the core library for numerical computations in Python. It provides support for multi-dimensional arrays and a wide range of mathematical operations. Compared to Python lists, NumPy arrays are faster, more memory-efficient, and allow advanced operations such as:
    1. Linear algebra and Fourier transforms.
    2. Random number generation.
    3. Statistical operations.
    4. Efficient handling of large datasets.
- ***Screen Brightness Control / Pycaw:*** These libraries were used to adjust screen brightness and control system volume directly through Python. They made our project more user-friendly by providing real-time control over system settings [6].
- ***Arduino:*** Arduino is an open-source hardware and software platform for building electronic projects. It consists of a microcontroller board and an IDE for programming it. In our project, Arduino was used as the hardware output device to receive commands from Python and perform actions. It is widely popular due to its simplicity, low cost, and strong community support, making it ideal for students and hobbyists.

3.2 Flow of the Software

The working of the software can be explained step by step:

1. **Start the App:** When we run the application, the GUI opens.
2. **Camera Capture:** The camera starts automatically and continuously captures frames.
3. **Gesture Detection:** MediaPipe processes each frame to identify hand landmarks.
4. **Finger Status:** The software checks which fingers are up or down using a simple logic.
5. **Gesture Matching:** Based on finger patterns, a gesture name is given like "Index Up", "Fist" etc.
6. **Action Mapping:** Each gesture is mapped to a specific action that we define in the settings.
7. **Command Execution:** According to the mapping, the software either sends a signal to Arduino or controls system functions like volume or brightness.
8. **Output Update:** The app shows the last detected gesture in the interface and updates the connected hardware.

3.3 Hand Gesture Detection

The most important part of the software is hand gesture detection. For this, we use Mediapipe Hands, which is a machine learning solution. Mediapipe can detect 21 landmarks of each hand such as the tips of fingers, joints, and wrist. With these landmarks, we can calculate whether a finger is up or down.

For example:

- If only the index finger is up → Gesture = **Index Up**
- If no fingers are up → Gesture = **Fist**
- If all five fingers are up → Gesture = **Palm**
- If thumb and index form a circle → Gesture = **OK**
- If index and middle fingers are up → Gesture = **Two Fingers**
- If only thumb is up → Gesture = **Thumbs Up**
- If thumb, index, and little finger is up → Gesture = **Yo**

3.4 Gesture to Action Mapping

Once the gesture is detected, the next step is to map it to an action. For this, we created a dictionary in Python where each gesture name is linked with an action and a parameter. This mapping is also saved in a JSON file so that it can be used next time without re-entering.

For example:

- Gesture "Fist" → Action: Toggle_Device → **Arduino OFF**
- Gesture "Palm" → Action: Volume_Set 50 → **System Volume 50%**
- Gesture "Index Up" → Action: Toggle Device → **Arduino ON**

4

3.5 Graphical User Interface (GUI)

We used Kivy to design a proper user interface. The GUI is divided into two main sections:

1. **Left Side (Camera Preview):**
    - Shows the live camera feed with hand landmarks drawn on it.
    - Buttons to Start/Stop camera, Hide camera window, and Test volume.
2. **Right Side (Gesture Settings):**
    - List of all gestures with their actions and parameters.
    - Buttons to add or delete gestures.
    - Options to save mappings.
    - Parameters like Detection Confidence, Max Hands, Camera Index, and Action Cooldown.
    - A label that shows the "Last Gesture" detected.

3.6 Arduino Communication

For communication with the Arduino, we used the pyserial library. The software automatically detects the Arduino port (COM port) and connects at a baud rate of 9600. Commands are sent in simple text like:

- "ON" → **Lamp ON**
- "OFF" → **Lamp OFF**
- "BRIGHTNESS:200" → **Set brightness level to 200**

On the Arduino side, a simple code is written to receive these commands and perform the required output.

3.7 Actions Performed by Gestures

The software allows us to perform different actions such as:

1. **Toggle Device:** Turns the Arduino output ON or OFF.
2. **Brightness Control:** Increases, decreases, or sets brightness using Arduino.
3. **Volume Control:** Changes system volume up or down.
4. **Hide Camera:** Stops displaying the live preview in the GUI.

3.8 Safety and Error Handling

The software is written in such a way that even if Arduino is not connected, the app still runs. It shows a warning “Arduino not connected” but continues to detect gestures. This avoids crashing and makes it more stable. Similarly, exceptions are handled for brightness or volume libraries if they are not available on the system.

Chapter 4-Hardware Setup

4.1 Components Used

The hardware setup is very simple and easy to understand. The main components used are:

- ***Arduino Board (Uno)*:** This is the main microcontroller that receives commands from the software and controls the output devices.
- ***Lamps*:** We used lamps to show visual output. Lamps light up according to the commands sent by the software via Arduino.
- ***Connecting Wires and Breadboard*:** These help in connecting all the components without soldering, making it easy to change the connections.
- ***USB Cable*:** Used to connect Arduino to the computer for communication.
- ***Power Supply:*** Arduino gets power from the USB connection of the computer, so no extra power supply is required.

These components together form a simple but effective hardware setup for our gesture-controlled system.

4.2 Arduino Setup

The Arduino board is programmed to receive commands from the software through the serial communication port (COM port). The commands are sent in simple text like:

- "ON" → **Turn LAMPS on**
- "OFF" → **Turn LAMPS off**
- "BRIGHTNESS:200" → **Set brightness level to 200**

Arduino reads these commands and performs the required output. For example, when the software detects the “Fist” gesture, it sends "OFF" to Arduino, and the lamp turns off. If it detects “Index Up,” it sends "ON", and the lamp turns on.

This communication is very fast and happens in real-time, so the lamp responds immediately to hand gestures.

4.3 Lamps Circuit

The lamps circuit is simple and easy to build. Each lamp is connected to one digital pin of Arduino through a resistor. The resistor limits the current to protect the lamp. The other end of the lamp is connected to the GND (ground) pin on the Arduino.

For example:

- Pin 8 → Resistor → LAMP → GND
- Pin 9 → Resistor → LAMP → GND

This setup allows us to control each lamp individually using the Arduino commands sent from the software.

4.4 Controlling Brightness

For brightness control, we use PWM (Pulse Width Modulation) pins on Arduino. PWM allows us to vary the intensity of the LAMPS by quickly turning it ON and OFF at different speeds.

- If we send "BRIGHTNESS:255" → **Lamp shines at Full Brightness.**
- If we send "BRIGHTNESS:128" → **Lamp shines at Half Brightness.**
- If we send "BRIGHTNESS:0" → **Lamp is Off**.

This simulates real-world applications like dimming lights or controlling motor speed using hand gestures.

4.5 Advantages of Arduino

Arduino is chosen for this project because it has many benefits:

1. Easy to Program: It uses a simple programming language similar to C/C++.
2. Affordable: Arduino boards are cheap and easily available.
3. Versatile: Can control lamp, motors, relays, and sensors.
4. Community Support: Many tutorials and examples are available online.
5. Reliable: Works well with PC software via serial communication.

Because of these features, Arduino is the perfect choice for gesture-controlled hardware systems.

4.6 Serial Communication

The hardware communicates with software through serial communication. Serial communication is a way of sending data one bit at a time over a single wire.

- The software sends commands through the USB cable.
- Arduino reads the commands using Serial.read() or Serial.readString() function.
- Based on the command, Arduino controls the lamp or other devices.

This simple method makes the hardware respond in real-time and ensures the system works smoothly.

4.7 Hardware Testing

Before connecting the software, we first test the hardware independently. This includes:

1. Connecting lamp to Arduino and testing ON/OFF manually.
2. Testing brightness control using simple Arduino code.
3. Checking serial communication by sending test messages from the PC.

Once the hardware works perfectly on its own, it can be integrated with the software for gesture control.

4.8 Real-Time Interaction

The combination of software and hardware allows real-time interaction. For example:

- When the user shows a “Palm” gesture → Software detects → Sends "ON" → Arduino turns lamp ON instantly.
- When the user shows a “Fist” gesture → Software detects → Sends "OFF" → lamp turns OFF.
- Brightness gestures like “Thumbs Up” can increase the lamp brightness gradually.

This demonstrates how software and hardware work together to create a smart gesture-controlled system.

Chapter 5-Coding

5.1 Import Section

At the beginning of the code, we import all the necessary modules required for different parts of the project. These include libraries for building the user interface, handling camera input, detecting hand gestures, managing threads, saving settings, and communicating with the Arduino. Each import corresponds to a specific feature of the system and ensures that the software runs smoothly, efficiently, and in real time.

```
from kivy.app import App
from kivy.lang import Builder
from kivy.clock import Clock, mainthread
from kivy.properties import DictProperty, BooleanProperty, StringProperty, NumericProperty
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.popup import Popup
from kivy.graphics.texture import Texture

import threading
import time
import json

# CV / Mediapipe imports
import cv2
import mediapipe as mp
import numpy as np

import serial
import serial.tools.list_ports
```

5.2 Arduino Port Detection and Connection

This section of the code automatically detects the COM port to which the Arduino is connected. The find_arduino_port() function scans all available serial ports and checks their descriptions for common identifiers like "Arduino", "CH340", or "USB-SERIAL". If a matching port is found, it returns the port name (e.g., COM4, /dev/ttyUSB0). If not, the program falls back to a default port, COM3. After detecting the correct port, the program attempts to establish a serial connection with the Arduino using a baud rate of 9600. A short delay (time.sleep(2)) is added to give the Arduino time to reset after the connection is opened. If the connection is successful, it prints the connected port name. If it fails, the Arduino object is set to None, and an error message is displayed, allowing the program to continue running without crashing.

```python
# Detect Arduino port
def find_arduino_port():
    ports = serial.tools.list_ports.comports()
    for p in ports:
        if "Arduino" in p.description or "CH340" in p.description or "USB-SERIAL" in p.description:
            return p.device
    return None

try:
    arduino_port = find_arduino_port() or 'COM3'  # Fallback if not auto-detected
    arduino = serial.Serial(arduino_port, 9600, timeout=1)
    time.sleep(2)  # Wait for Arduino reset after opening port
    print(f"Connected to Arduino on {arduino_port}")
except Exception as e:
    arduino = None
    print("Arduino not connected:", e)
```

5.3 Sending Commands to Arduino

The send_arduino_command(cmd) function is responsible for sending text-based commands from the software to the Arduino over a serial connection. Before sending, it checks if the Arduino is connected and the serial port is open. If everything is fine, it appends a newline character (\n) to the command (to indicate the end of the message) and encodes it into bytes using UTF-8 format — which is required for serial transmission. The encoded command is then written to the Arduino. If any error occurs during this process, it is caught and displayed, preventing the program from crashing. This function ensures safe and reliable communication with the Arduino for triggering physical actions like switching devices on or off.

```python
def send_arduino_command(cmd):
    if arduino and arduino.is_open:
        try:
            arduino.write((cmd + '\n').encode('utf-8'))
            print(f"Sent to Arduino: {cmd}")
        except Exception as e:
            print("Failed to write to Arduino:", e)
```

5.4 System Feature Imports (Brightness & Volume Control)

This code checks whether the system supports volume control using the **pycaw** library on Windows. It tries to import the necessary modules for accessing and controlling audio devices; if successful, it sets a flag _PYCAW_AVAILABLE to True. If the import fails (e.g. if the required libraries aren't installed or the platform isn't Windows), it sets the flag to False. This allows the program to conditionally enable or disable volume control features based on system compatibility.

```
try:
    # pycaw for Windows volume control
    from ctypes import POINTER, cast
    from comtypes import CLSCTX_ALL
    from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume
    _PYCAW_AVAILABLE = True
except Exception:
    _PYCAW_AVAILABLE = False
```

5.5 Function to Detect Raised Fingers from Hand Landmarks

This function analyzes hand landmark data to determine which fingers are raised. It uses the position of fingertip landmarks to detect the status (up or down) of each finger. The thumb detection is based on horizontal (x-axis) position and depends on whether the hand is labeled as "Right" or not. The other four fingers (index to pinky) are detected using vertical (y-axis) positions. It returns a list of five values (1 or 0), where 1 means the finger is up and 0 means it's down.

```
def fingers_up(hand_landmarks, hand_label):
    fingers = []
    tip_ids = [4,8,12,16,20]
    try:
        if hand_label == 'Right':
            fingers.append(1 if hand_landmarks.landmark[tip_ids[0]].x < hand_landmarks.landmark[tip_ids[0]-1].x else
            0)
        else:
            fingers.append(1 if hand_landmarks.landmark[tip_ids[0]].x > hand_landmarks.landmark[tip_ids[0]-1].x else
            0)
    except Exception:
        fingers.append(0)
    for id in range(1,5):
        try:
            fingers.append(1 if hand_landmarks.landmark[tip_ids[id]].y < hand_landmarks.landmark[tip_ids[id]-2].y
            else 0)
        except Exception:
            fingers.append(0)
    return fingers
```

5.6 Function to Match Finger Pattern to Predefined Gestures

This function takes a list representing the state of five fingers (1 for up, 0 for down) and matches it to a predefined gesture. It checks for specific finger combinations corresponding to common gestures like "Fist", "Palm", "Thumbs Up", etc. If a match is found, it returns the name of the gesture; otherwise, it returns None. This is typically used after detecting finger positions to identify user gestures.

```python
def match_gesture(fingers):
    if fingers == [0,1,0,0,0]:
        return 'Index Up'
    elif fingers == [0,0,0,0,0]:
        return 'Fist'
    elif fingers == [1,1,1,1,1]:
        return 'Palm'
    elif fingers == [1,1,0,0,0]:
        return 'OK'
    elif fingers == [1,1,0,0,1]:
        return 'Yo!'
    elif fingers == [1,0,0,0,0]:
        return 'Thumbs Up'
    elif fingers == [0,1,1,0,0]:
        return 'Two Fingers'
    else:
        return None
```

5.7 Function to Control Volume or Brightness via Arduino Commands

This function performs volume or brightness adjustments by sending commands to an Arduino device. It takes an action (like "volume_up" or "brightness_set") and an optional numeric param (defaulting to 10 if not provided). For increase actions, it sends "ON", and for decrease actions, it sends "OFF". For set actions, it converts the given percentage into a 0–255 scale and sends a brightness command in the format "BRIGHTNESS:<value>". Errors during execution are caught and logged to avoid crashes.

```python
def perform_volume_action(action, param):
    try:
        val = float(param) if param else 10.0
        if action in ['volume_up', 'brightness_up']:
            send_arduino_command("ON")
        elif action in ['volume_down', 'brightness_down']:
            send_arduino_command("OFF")
        elif action in ['volume_set', 'brightness_set']:
            brightness = int(val * 255 / 100)
            send_arduino_command(f"BRIGHTNESS:{brightness}")
    except Exception as e:
        print('Error performing action:', e)
```

5.8 Function to Adjust Screen Brightness Using System Brightness Control

This function changes the screen brightness using the sbc (screen brightness control) library, if available. Based on the action parameter, it can increase, decrease, or set the brightness to a specific level. If no value is provided, it defaults to adjusting by 5% or setting brightness to 50%. The function ensures brightness stays within the 0–100% range. Any errors during the process are caught and printed to avoid crashes. If the sbc library is not available, it informs the user and skips execution.

```
def perform_brightness_action(action, param):
    if not sbc:
        print('Brightness library not available')
        return
    try:
        if action == 'brightness_up':
            cur = sbc.get_brightness()
            if isinstance(cur, list):
                cur = cur[0]
            delta = float(param) if param else 5.0
            sbc.set_brightness(min(100, cur + delta))
        elif action == 'brightness_down':
            cur = sbc.get_brightness()
            if isinstance(cur, list):
                cur = cur[0]
            delta = float(param) if param else 5.0
            sbc.set_brightness(max(0, cur - delta))
        elif action == 'brightness_set':
            v = float(param) if param else 50.0
            sbc.set_brightness(max(0,min(100,v)))
    except Exception as e:
        print('Error performing brightness action:', e)
```

5.9 MainUI Class for Gesture-Controlled Interface Configuration

The MainUI class is the main layout and control center for the gesture-based application interface. Inheriting from BoxLayout, it defines various properties that manage both the application's state and user-configurable settings. The mappings dictionary stores gesture-to-action mappings created by the user. Flags like cam_running and cv_window track whether the camera and OpenCV preview window are active. The cam_toggle_text controls the label on the camera toggle button. Parameters such as detection_confidence, max_hands, camera_index, and action_cooldown allow

users to adjust the gesture detection behavior—like setting how confident the detection should be, how many hands to track, which camera to use, and how much time to wait before accepting another gesture. Lastly, last_gesture holds the name of the most recently detected gesture, which is displayed in the UI. These properties are all bound to the interface and update in real time as the app runs.

```
class MainUI(BoxLayout):
    mappings = DictProperty({})
    cam_running = BooleanProperty(True)
    cv_window = BooleanProperty(False)
    cam_toggle_text = StringProperty('Stop Camera')
    detection_confidence = NumericProperty(0.75)
    max_hands = NumericProperty(2)
    camera_index = NumericProperty(0)
    action_cooldown = NumericProperty(1.0)
    last_gesture = StringProperty('None')
```

5.10 Utility Methods for Managing Gesture Mappings and Camera Controls

These methods provide core functionality for managing gesture-action mappings and toggling camera behavior within the app. The update_mapping method adds or updates a gesture mapping in the mappings dictionary using the provided gesture name, action, and parameter. After updating, it calls populate_gesture_rows() to refresh the UI and reflect changes. Similarly, delete_mapping removes a mapping by gesture name, if it exists, and then updates the interface accordingly.

The toggle_camera method switches the camera on or off based on the current state. It flips the cam_running flag and updates the cam_toggle_text to reflect the new button label ("Start Camera" or "Stop Camera"). Depending on the new state, it either starts or stops the camera loop.

Lastly, toggle_cv_window enables or disables the OpenCV preview window (cv2.imshow). If disabling, it attempts to close any open OpenCV windows gracefully using cv2.destroyAllWindows(). Together, these methods handle user interaction with gesture mappings and camera settings, maintaining a responsive and intuitive UI.

```
def update_mapping(self, name, action, param):
    self.mappings[name] = {'action': action, 'param': param}
    self.populate_gesture_rows()

def delete_mapping(self, name):
    if name in self.mappings:
        self.mappings.pop(name)
    self.populate_gesture_rows()

def toggle_camera(self):
    self.cam_running = not self.cam_running
    self.cam_toggle_text = 'Start Camera' if not self.cam_running else 'Stop Camera'
    if self.cam_running:
        self.start_camera()
    else:
        self.stop_camera()

def toggle_cv_window(self):
    self.cv_window = not self.cv_window
    if not self.cv_window:
        try:
            cv2.destroyAllWindows()
        except Exception:
            pass
```

5.11 Control Methods for Gesture Detection Settings and Camera Handling

This set of methods belongs to the MainUI class and provides essential controls for updating detection settings, testing functionality, and managing the camera thread. The update_confidence method sets the minimum confidence level required for gesture detection, while update_max_hands updates how many hands the system should track (usually 1 or 2). The test_volume method sends a test volume-up command with a default increment of 10 using the perform_volume_action function, allowing users to verify volume control functionality. The start_camera method initializes and starts a new background thread to run the camera loop (_camera_loop), unless it's already running. It uses a threading event (_stop) to manage thread control. Conversely, stop_camera sets the stop flag, which signals the camera loop to terminate, effectively stopping the video feed. Together, these methods provide a clean interface for adjusting settings and handling camera input in a responsive, threaded manner.

```
def update_confidence(self, val):
    self.detection_confidence = val

def update_max_hands(self, n):
    self.max_hands = n

def test_volume(self):
    perform_volume_action('volume_up', '10')

def start_camera(self):
    self._stop.clear()
    if self.capture_thread and self.capture_thread.is_alive():
        return
    self.capture_thread = threading.Thread(target=self._camera_loop, daemon=True)
    self.capture_thread.start()

def stop_camera(self):
    self._stop.set()
```

5.12 Camera Loop for Real-Time Gesture Detection and Display

The _camera_loop method is a background thread responsible for handling real-time video capture, gesture detection, and UI updates in the application. It initializes MediaPipe's hand tracking module with the user-defined max_hands and detection_confidence settings. The method opens the selected camera (based on the input from the UI) and sets the resolution. In a continuous loop (until a stop flag is triggered), it reads frames from the camera, flips them horizontally (for mirror view), and processes them using MediaPipe to detect hand landmarks. If hands are detected, it draws the landmarks and checks which fingers are raised using the fingers_up function. The finger pattern is matched to a predefined gesture using match_gesture. If a valid gesture is recognized and enough time has passed since the last action (based on cooldown), it triggers on_gesture_detected. The loop optionally displays the frame in an OpenCV window if enabled by the user, allowing manual viewing/debugging. It also updates the Kivy UI by converting the current frame to a texture and applying it to the image widget in the interface. Once stopped, the method releases the camera and safely closes any OpenCV windows. This function effectively bridges the gap between the gesture recognition backend and the live UI, enabling interactive, responsive gesture control.

```
def _camera_loop(self):
    mp_hands = mp.solutions.hands
    mp_drawing = mp.solutions.drawing_utils
    hands = mp_hands.Hands(max_num_hands=self.max_hands, min_detection_confidence=self.detection_confidence)
    cap = cv2.VideoCapture(int(self.ids.cam_idx.text) if self.ids.cam_idx.text.isdigit() else 0)
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    while not self._stop.is_set():
        success, frame = cap.read()
        if not success:
            time.sleep(0.1)
            continue
        frame = cv2.flip(frame, 1)
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        rgb_frame.flags.writeable = False
        result = hands.process(rgb_frame)
        rgb_frame.flags.writeable = True

        # draw landmarks
        if result.multi_hand_landmarks and result.multi_handedness:
            for i, (hand_landmarks, hand_handedness) in enumerate(zip(result.multi_hand_landmarks, result.
            multi_handedness)):
                hand_label = hand_handedness.classification[0].label
                mp_drawing.draw_landmarks(frame, hand_landmarks, mp_hands.HAND_CONNECTIONS)
                fingers = fingers_up(hand_landmarks, hand_label)
                gesture_name = match_gesture(fingers)
                if gesture_name:
                    now = time.time()
```

```
                if gesture_name:
                    now = time.time()
                    if now - self._last_action_time > float(self.ids.cooldown.text):
                        self._last_action_time = now
                        self.on_gesture_detected(gesture_name)

        # Optionally show cv2 window
        if self.cv_window:
            cv2.imshow('Camera (cv2)', frame)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                self._stop.set()

        # Update Kivy image texture
        buf = cv2.flip(frame, 0).tobytes()
        h, w = frame.shape[:2]
        texture = Texture.create(size=(w, h), colorfmt='bgr')
        texture.blit_buffer(buf, colorfmt='bgr', bufferfmt='ubyte')
        self.update_texture(texture)

    cap.release()
    try:
        cv2.destroyAllWindows()
    except Exception:
        pass
    hands.close()
```

5.13 KV Definition for GestureRow Widget

This KV block defines the layout for a custom widget called GestureRow, which is used to represent a single gesture-action mapping in the user interface. The layout is a horizontal BoxLayout with a fixed height and spacing between elements. It includes three user input fields: a TextInput for the gesture name (gname), a Spinner for selecting an action (action) from a predefined list (e.g., volume or brightness controls), and another TextInput for an optional parameter (param) related to the action. Additionally, there are two buttons: a **"Save"** button, which calls the on_save() method on the GestureRow with the current values of the fields, and a **"×" (delete)** button, which calls on_delete() to remove the gesture row. This component allows users to define and manage gesture bindings in a clean and structured way within the app.

```
KV = '''
<GestureRow@BoxLayout>:
    orientation: 'horizontal'
    size_hint_y: None
    height: '36dp'
    spacing: 6
    gname: ''
    action: ''
    param: ''
    TextInput:
        id: gname
        text: root.gname
        multiline: False
    Spinner:
        id: action
        text: root.action
        values: ['toggle_device','volume_up','volume_down','volume_set','brightness_up','brightness_down',
        'brightness_set','hide_camera']
        size_hint_x: .5
    TextInput:
        id: param
        text: root.param
        multiline: False
        size_hint_x: .5
    Button:
        text: 'Save'
        size_hint_x: .3
        on_release: root.on_save(gname.text, action.text, param.text)
    Button:
        text: '×'
        size_hint_x: None
        width: '40dp'
```

5.14 KV Layout for MainUI – Camera Preview and Control Panel

This section of the KV language defines the **left panel** of the MainUI layout, which focuses on camera preview and control functions. It uses a vertical BoxLayout occupying 60% of the horizontal space (size_hint_x: .6). At the top, a Label displays the title **"Camera Preview"**. Below that, an Image widget (id: cam_view) is used to show the live camera feed. The image is set to stretch while keeping its aspect ratio and fills most of the vertical space (size_hint_y: .8).

At the bottom of the panel, a horizontal BoxLayout contains three buttons:

1. A **camera toggle button**, which displays dynamic text (root.cam_toggle_text) and calls root.toggle_camera() when clicked to start or stop the camera.
2. A button labeled **"Hide Camera Window (cv2.imshow)"**, which toggles the OpenCV window display using root.toggle_cv_window().
3. A **"Test Volume"** button, which triggers a volume test by calling root.test_volume().

This layout provides users with essential camera controls and visual feedback, allowing them to preview the video feed and interact with the system directly.

```
<MainUI>:


    orientation: 'horizontal'
    padding: 8
    spacing: 8

    BoxLayout:
        orientation: 'vertical'
        size_hint_x: .6

        Label:
            text: 'Camera Preview'
            size_hint_y: None
            height: '30dp'
        Image:
            id: cam_view
            allow_stretch: True
            keep_ratio: True
            size_hint_y: .8
        BoxLayout:
            size_hint_y: None
            height: '36dp'
            spacing: 6
            Button:
                text: root.cam_toggle_text
                on_release: root.toggle_camera()
            Button:
                text: 'Hide Camera Window (cv2.imshow)'
                on_release: root.toggle_cv_window()
            Button:
                text: 'Test Volume'
```

5.15 Arduino Code to Control Lamp Brightness via Serial Commands

This Arduino sketch controls a lamp connected to pin 10 (a PWM-capable pin) using commands received over the serial interface. It initializes serial communication at 9600 baud and sets pin 10 as an output with the lamp initially off.

In the main loop, it reads incoming serial data character-by-character until it encounters a newline (\n). It then processes the input command:

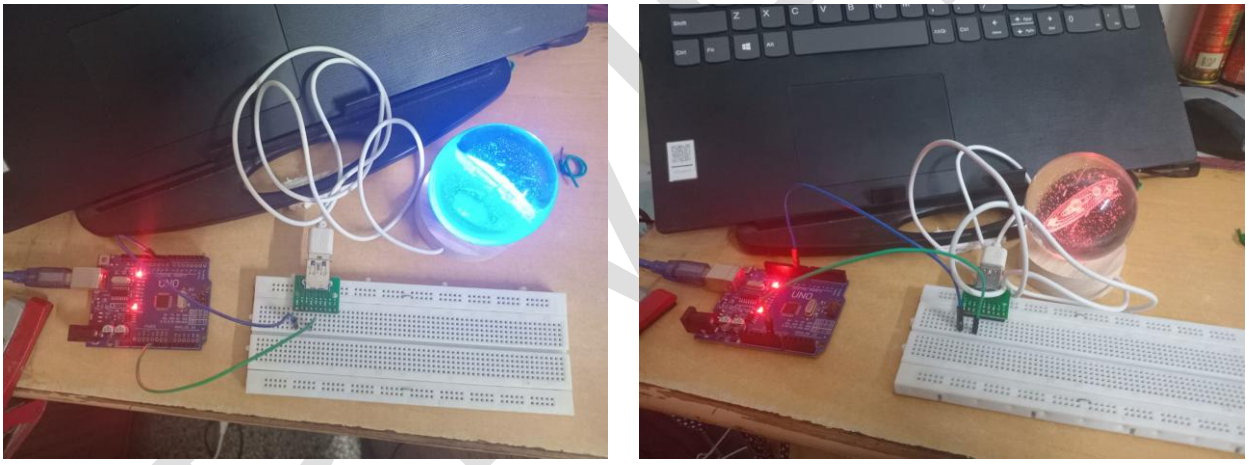
- "ON" sets the lamp to full brightness (PWM value 255).
- "OFF" turns the lamp off (PWM value 0).
- Commands starting with "BRIGHTNESS:" followed by a number set the lamp brightness to that PWM value, constrained between 0 and 255.

This enables dynamic control of the lamp's brightness through simple serial messages.

```
#define LAMP_PIN 10  // PWM-capable pin

void setup() {
  Serial.begin(9600);
  pinMode(LAMP_PIN, OUTPUT);
  analogWrite(LAMP_PIN, 0); // Lamp off at start
}

void loop() {
  static String input = "";

  while (Serial.available() > 0) {
    char c = Serial.read();
    if (c == '\n') {
      input.trim();

      if (input == "ON") {
        analogWrite(LAMP_PIN, 255);
      } else if (input == "OFF") {
        analogWrite(LAMP_PIN, 0);
      } else if (input.startsWith("BRIGHTNESS:")) {
        int value = input.substring(11).toInt();
        value = constrain(value, 0, 255);
        analogWrite(LAMP_PIN, value);
      }

      input = "";  // Clear for next command
    } else {
      input += c;
    }
  }
}
```

Chapter 6-Output

Smart-Sense Living: An Adaptive IoT Control Prototype-The system has been carefully designed to detect hand gestures using a camera, process them through software, and then control an external device — in our case, a lamp — via an Arduino board. When we launch the application, the camera begins capturing live video. The software developed using Python, OpenCV, and MediaPipe, constantly monitors the frame for the presence of a hand and identifies the specific gesture being made. These gestures are predefined in the program's code. For example, when an open palm is shown to the camera, the software recognizes this as the command "ON" and instantly transmits it to the Arduino through a serial connection. The Arduino, in turn, activates the lamp, lighting it up in response. Similarly, showing a closed fist is recognized as the "OFF" command, which causes the lamp to turn off immediately as shown in Fig. 6.1.



(a) **(b)**

Fig. 6.1 (a) "Palm" gesture → Lamp ON, (b) "Fist" gesture → Lamp OFF.

For brightness control, we incorporated additional gestures such as a thumbs-up and thumbs-down as shown in Fig. 6.2. These allow the user to adjust the lamp's brightness incrementally. The software translates these gestures into precise commands that are sent to the Arduino, which then modifies the brightness levels accordingly. This dynamic response creates an interactive environment where even a subtle change in hand motion produces a visible reaction. The responsiveness of the system demonstrates not only the efficiency of the code. Overall, the results of SmartSenseLiving confirm that gesture-based control systems can be implemented successfully

with accessible tools and technologies. The photographs of the lamp in action clearly capture how well the system performs under real-world conditions.

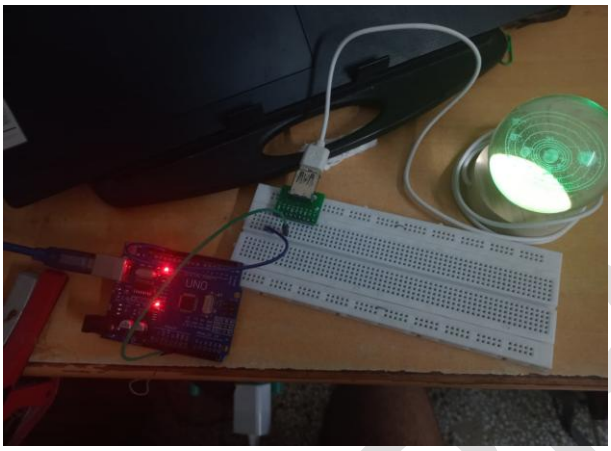


Fig. 6.2 "Thumbs Up" can increase the lamp brightness gradually.

Chapter 7-Conclusion

In this project, we created a gesture-control Lamps system that combines software and hardware to perform real-world actions. The main goal was to show how human gestures can be detected by a computer and then used to control devices like lamps or other electronics. This project is an example of human-computer interaction and smart home technology, which is becoming very important today.

For the software, we used Python and Kivy to make a user-friendly interface. We also used MediaPipe to detect hand gestures in real time. The software can recognize gestures like “Fist,” “Palm,” and “Thumbs Up.” When a gesture is detected, the software sends commands to the hardware via Arduino or Socket.IO. This demonstrates how software acts as the brain of the system, making decisions and sending instructions to hardware.

For the hardware, we used Arduino as the main controller. Arduino receives commands from the software and controls output devices like lamps. Features like brightness control using PWM pins allow smooth and adjustable control. The hardware setup is simple, inexpensive, and safe, making it suitable for beginners and students.

A key achievement is the real-time interaction between gestures and output. For example, showing a “Fist” turns off the lamps, and “Index Up” turns it on. This demonstrates practical applications of gesture recognition, especially in situations where touch is difficult, unhygienic, or for people with special abilities.

Through this project, we learned about Python programming, GUI development with Kivy, real-time data handling, and Arduino control. It also improved our problem-solving skills and understanding of electronics. Although the project works well, future improvements could include controlling more appliances, adding more gestures, and adapting the system for darker environments.

Overall, this project is a successful beginner-friendly demonstration of software-hardware integration, human-computer interaction, and real-time gesture control. It provides a strong foundation for more advanced projects in smart home automation and robotics.

Chapter 8-Summary and Future Scope

This project presents a gesture-controlled lamp system that integrates software and hardware to perform real-world actions through human gestures. Using Python, Kivy, and MediaPipe, the system detects gestures such as Fist, Palm, and Thumbs Up in real time and communicates commands to Arduino via Socket.IO. The Arduino then controls the lamp's operation, including brightness adjustment using PWM pins.

The project demonstrates effective human-computer interaction and serves as a beginner-friendly example of smart home automation. It showcases how gestures can control devices without physical touch—useful in hygienic or accessibility-sensitive situations. Through this work, we gained practical experience in Python programming, GUI design, real-time processing, and hardware interfacing. Future improvements may include more gestures, support for multiple appliances, and better performance in low-light conditions.

Overall, the system successfully demonstrates real-time gesture recognition and software-hardware integration, forming a solid foundation for advanced IoT and robotics applications.

The gesture-controlled system that we created can be improved and expanded in many ways in the future.

- ***Integration with Smart Home Devices***: The system can be connected to smart home appliances like fans, lights, air conditioners, and TVs. This will allow users to control multiple devices in their house just with gestures.
- ***Voice and Gesture Combination***: The system can be combined with voice recognition technology so that gestures and voice commands can be used together. This will make controlling devices even easier.
- ***Mobile Application Integration***: Currently, the system works on a PC with Kivy. In the future, it can be fully integrated into a mobile app so that users can control devices from anywhere using their smartphones.
- ***Industrial Applications***: This technology can be applied in factories or workplaces where touch-free control is important. Workers can control machines without physically touching them, reducing the risk of accidents.

- ***Healthcare Applications***: Gesture control can be very useful in hospitals. Doctors or patients can control devices like lights or screens without touching anything, which is hygienic and reduces infection risk.

REFERENCES

[1] A. Dix, J. Finlay, G. D. Abowd, and R. Beale, Eds., *Human–Computer Interaction*, Third Edit. Pearson Education Limited 2004 The.

[2] S. Mitra and T. Acharya, "Gesture recognition: A survey," *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, vol. 37, no. 3, pp. 311–324, May 2007, doi: 10.1109/TSMCC.2007.893280.

[3] "OpenCV: OpenCV modules." Accessed: Oct. 07, 2025. [Online]. Available: https://docs.opencv.org/4.x/

[4] "MediaPipe — MediaPipe v0.7.5 documentation." Accessed: Oct. 10, 2025. [Online]. Available: https://mediapipe.readthedocs.io/en/latest/

[5] "Welcome to Kivy — Kivy 2.3.1 documentation." Accessed: Oct. 10, 2025. [Online]. Available: https://kivy.org/doc/stable/

[6] "PyCaw — PyCaw latest documentation." Accessed: Oct. 07, 2025. [Online]. Available: https://pycaw.readthedocs.io/en/latest/