1. INTRODUCTION

In the rapidly evolving landscape of human-computer interaction (HCI), the integration of computer vision technologies has opened new frontiers in accessibility and user interface design. This project presents a Gesture Controlled Virtual Mouse that leverages the capabilities of the MediaPipe library for hand tracking and gesture recognition. The system provides users with an innovative means of interacting with their computers, transcending traditional mouse and keyboard inputs. By harnessing the power of a standard webcam, the virtual mouse interprets hand gestures, enabling intuitive control over the system. This project explores the fusion of artificial intelligence, computer vision, and user-friendly interfaces to create a seamless and efficient hands-on computing experience. Through the following documentation, we delve into the key components, functionalities, and implementation details of this cutting-edge virtual mouse system.

1.1 Background

Human-Computer Interaction (HCI) has witnessed a paradigm shift with the integration of computer vision technologies into daily computing experiences. Traditional input devices like mice and keyboards, while effective, may pose limitations in certain contexts. The emergence of hand tracking and gesture recognition technologies has spurred the development of novel HCI solutions, offering users more natural and immersive ways to engage with their devices. This project draws inspiration from this technological evolution, aiming to create a Gesture Controlled Virtual Mouse. The foundation of this endeavor lies in the MediaPipe library, a robust framework for hand tracking that empowers the system to interpret intricate hand movements and gestures. By tapping into the potential of artificial intelligence and computer vision, this project envisions a future where users can seamlessly navigate and interact with their computers through intuitive hand gestures, transcending the conventional boundaries of input devices. In exploring this background, we delve into the motivations and aspirations that drive the development of a virtual mouse system, poised at the intersection of human interaction and cutting-edge technology.

1.2 Motivation (Problem Statement)

Current human-computer interaction (HCI) relies heavily on traditional input devices like mice and keyboards. While widely used, these tools can hinder users in specific scenarios. People with physical limitations may struggle with conventional input methods, and others might desire a more intuitive, hands-free approach. The growing presence of webcams in computers offers an opportunity to address these limitations. This project tackles the need for alternative input methods by leveraging hand tracking and gesture recognition through the MediaPipe library. The core problem lies in offering users a novel way to interact with their computers, surpassing the constraints of traditional devices. By removing the dependence on physical peripherals, the virtual mouse system aims to improve accessibility, reduce physical strain, and cater to diverse user preferences. Furthermore, this project seeks to explore the broader integration of artificial intelligence (AI) and computer vision into everyday computing tasks. The goal is to push the boundaries of HCI and create a more inclusive digital experience. This project strives not only to solve the limitations of traditional input devices but also to pave the way for advancements in gesture-based computing.

1.3 Existing System

As of the current state of technology, the predominant means of interacting with computers involve traditional input devices such as mice, keyboards, and touchpads. While these devices have proven to be reliable and widely adopted, they come with certain limitations, especially in addressing diverse user needs and preferences. Users with physical disabilities or those seeking alternative input methods may find the existing system less accommodating. Additionally, the reliance on physical peripherals can lead to issues of repetitive strain and discomfort during prolonged usage. Recognizing these limitations, there is a growing need to explore innovative solutions that enhance accessibility, minimize physical strain, and offer a more natural means of interaction.

1.3.1 Overview Of Existing System And Its Disadvantages

Limited Accessibility: Traditional input devices may not be fully accessible to users with physical disabilities, limiting their ability to interact with computers effectively.

Example: A user with a mobility impairment may find it challenging to use a standard mouse or keyboard, limiting their ability to navigate and interact with the computer effectively.

Repetitive Strain: Extended use of mice and keyboards can result in repetitive strain injuries and discomfort, impacting the overall user experience and health.

Example: Individuals who spend extended hours on computer-related tasks, such as data entry or design work, may experience repetitive strain injuries like carpal tunnel syndrome or tendonitis due to the repetitive movements required by traditional input devices.

Lack of Intuitive Interaction: The current system may lack the intuitiveness and natural interaction that users seek, especially in scenarios where a more hands-free approach is desired.

Example: Individuals who spend extended hours on computer-related tasks, such as data entry or design work, may experience repetitive strain injuries like carpal tunnel syndrome or tendonitis due to the repetitive movements required by traditional input devices.

Dependency on External Hardware: Existing systems require users to have specific external hardware devices, such as mice or touchpads, for interaction. This dependency can be restrictive in certain contexts.

Example: A user working on a laptop without an external mouse or touchpad may find it inconvenient to carry additional peripherals, especially when needing to work in various environments where external hardware may not be readily available.

One-size-fits-all Approach: The traditional system follows a one-size-fits-all approach, potentially neglecting the diverse preferences and needs of users who may benefit from alternative input methods.

Example: Users with unique preferences or physical conditions may feel underserved by the standardized design of traditional input devices. For instance, someone preferring a gesture-based interaction model may find the conventional setup less accommodating.

1.4 Proposed system

The Virtual Mouse Hand Recognition application utilizes a straightforward neural network on the finger, eliminating the need for additional hardware to control the cursor through simple gestures and hand movements. This functionality is achieved through vision-based hand gesture recognition, utilizing input from a webcam.

The proposed system employs a webcam for capturing images or video frames. To capture these frames, the OpenCV library, which is part of the Python programming language, is utilized. The webcam initiates video capture, and OpenCV creates an object for video capture. The frames captured by the webcam are then passed to the AI-based virtual system for processing.

1.4.1 Overview of proposed system and its limitations

Accuracy: While the system offers a high degree of accuracy, it may not always produce perfect results, especially with sarcastic or nuanced language.

User Interface: The user interface may require further refinement to ensure optimal usability for users with limited technical expertise.

Compatibility: The code and modules should be compatible in the system to make use of gestures.

Low Brightness: Within low light conditions, gestures are not captured by the webcam.

Performance: Real time Mouse operation depends upon the performance of the system. For instance, If the performance if the system is low then mouse operations can be delayed respectively.

1.5 Aim and purpose

Aim:

The primary aim of this project is to develop Gesture Controlled Virtual Mouse that redefines human-computer interaction through the integration of computer vision and gesture recognition technologies. The project seeks to offer users an innovative and accessible alternative to traditional input devices, allowing them to control and navigate their computers using intuitive hand gestures captured by a standard webcam.

Purpose:

Enhanced Accessibility: The project aims to address accessibility challenges faced by users with physical disabilities or those seeking alternative input methods. By leveraging computer vision, it strives to create a system that accommodates a diverse range of users, fostering inclusivity in digital interactions.

Intuitive and Natural Interaction: The purpose is to provide a more natural and intuitive means of interacting with computers. By recognizing and interpreting hand gestures in real-time, the virtual mouse system seeks to offer a user-friendly experience that goes beyond the constraints of traditional input devices.

Reduced Physical Strain: The project endeavors to mitigate the physical strain associated with prolonged use of mice and keyboards. Through hands-free interaction, users can potentially reduce repetitive strain injuries, contributing to a healthier and more comfortable computing experience.

Exploration of Advanced HCI: The purpose extends to exploring advanced Human-Computer Interaction (HCI) techniques by integrating artificial intelligence and computer vision. The project envisions a future where computers can adapt to user gestures, creating a foundation for more sophisticated and personalized interactions.

Innovation in Gesture-Based Computing: The purpose is to contribute to the evolution of HCI paradigms, allowing users to control their computers seamlessly through gestures, opening up possibilities for diverse applications and user scenarios.

1.6 Scope of the project

The scope of the Gesture Controlled Virtual Mouse encompasses several key dimensions, each contributing to its overarching goal of revolutionizing human-computer interaction. At its core, the project aims to develop a robust and versatile system capable of interpreting hand gestures captured by a standard webcam in real-time. This includes the implementation of advanced computer vision algorithms for hand tracking and gesture recognition, as well as the integration of artificial intelligence techniques for interpreting and responding to user inputs.

The project's scope extends beyond mere gesture recognition to encompass various functionalities and features aimed at enhancing user experience and accessibility. This includes functionalities such as cursor movement, clicking, scrolling, and potentially more complex actions like zooming or rotating, all performed through intuitive hand gestures. Additionally, the system may incorporate user preferences and customization options to tailor the interaction experience to individual needs.

In terms of hardware requirements, the project aims to maintain a reasonable level of accessibility by leveraging widely available resources such as standard webcams and computing devices. This ensures that users can readily adopt and utilize the virtual mouse system without the need for specialized equipment.

Furthermore, the project's scope encompasses usability and user interface considerations to ensure that the system is intuitive, user-friendly, and easy to navigate. While the primary focus of the project is on developing a functional prototype of the virtual mouse system, the scope also includes potential avenues for future expansion and refinement. This may involve optimization of algorithms for improved performance, integration with other software applications or platforms, and exploration of additional functionalities based on user feedback and emerging technologies

1.7 Objectives

Implement Hand Tracking: Develop algorithms to accurately track the movements and positions of the user's hands in real-time using computer vision techniques.

Gesture Recognition: Design and implement gesture recognition algorithms to interpret hand gestures captured by the webcam and translate them into meaningful commands for controlling the virtual mouse.

Cursor Control: Enable the virtual mouse system to control the movement of the cursor on the computer screen based on the detected hand gestures, allowing users to navigate the interface seamlessly.

Clicking and Selection: Implement functionality to recognize gestures indicating clicking or selecting actions, enabling users to interact with on-screen elements such as buttons, links, and icons.

Scrolling: Develop mechanisms to detect gestures representing scrolling motions, facilitating smooth vertical and horizontal scrolling within windows and documents.

Usability and Accessibility: Ensure that the virtual mouse system is user-friendly and accessible to a wide range of users, including those with physical disabilities or mobility impairments.

Performance Optimization: Optimize the performance of the system to minimize latency and ensure smooth and responsive interaction with the computer interface.

Error Handling: Implement robust error handling mechanisms to detect and recover from potential issues such as occlusions, lighting variations, or false detections, ensuring reliable operation under diverse conditions.

2. SYSTEM REQUIREMENT SPECIFICATIONS

2.1 Purpose of the System

The Gesture Controlled Virtual Mouse aims to redefine human-computer interaction by offering users an innovative and intuitive means of controlling their computers. At its core, the system is designed to provide an alternative to traditional input devices such as mice and keyboards, leveraging computer vision and artificial intelligence technologies to interpret hand gestures captured by a standard webcam. The primary purpose of the system is to enhance accessibility and usability for users, particularly those with physical disabilities or mobility impairments, by offering a hands-free method of interaction with computer interfaces.

By recognizing and interpreting hand gestures in Gesture Controlled Virtual Mouse enables users to navigate the computer interface, perform various actions such as clicking, scrolling, and selecting, and customize their interaction experience according to their preferences. Furthermore, the system aims to reduce the physical strain associated with prolonged use of traditional input devices, thereby promoting user comfort and well-being during computer-related tasks.

Beyond accessibility and usability enhancements, the virtual mouse system also fosters exploration and innovation in the field of human-computer interaction. By pushing the boundaries of gesture-based computing, the system opens up new possibilities for natural and intuitive interactions with computers, potentially paving the way for future advancements in interface design and user experience.

2.2 Feasibility analysis

By conducting a comprehensive feasibility analysis considering technical, economic, and operational factors, stakeholders can make informed decisions regarding the implementation of Gesture Controlled Virtual Mouse.

A feasibility analysis is essential to assess the viability of implementing the Gesture Controlled Virtual Mouse. Let's delve into the feasibility of this project from various perspectives

1. **Technical Feasibility**:

Availability of Computer Vision Algorithms: Numerous computer vision algorithms, such as those provided by OpenCV and MediaPipe, are readily available for hand tracking and gesture recognition. These algorithms have been extensively tested and proven effective in real-world applications.

Compatibility of Hardware Components: The system relies on standard hardware components like webcams, which are widely available and compatible with most computing devices. For instance, Logitech C920 HD Pro Webcam is commonly used for such applications due to its high resolution and frame rate capabilities.

Adequacy of Software Resources: The availability of programming languages like Python and libraries such as PyAutoGUI, OpenCV, and MediaPipe provide ample resources for developing the system. Additionally, frameworks like TensorFlow and PyTorch offer advanced machine learning capabilities for enhancing gesture recognition accuracy.

Real-time Examples: Companies like Google have implemented similar technologies in products like Google Pixel smartphones, where gesture-based interactions allow users to control various functions without touching the screen. This demonstrates the technical feasibility of Gesture Controlled Virtual Mouse.

2. **Economic Feasibility**:

Cost Considerations: The cost of implementing the Gesture Controlled Virtual Mouse primarily involves hardware, software, development, and testing expenses. However, the use of standard webcams and open-source software libraries helps keep costs relatively low compared to developing proprietary solutions.

Potential Return on Investment: The system can lead to improved accessibility and user experience, potentially resulting in increased productivity and user satisfaction. For example, integrating such technology into smart home devices can enhance user interaction and differentiate products in the market.

Real-time Examples: Companies like Microsoft have incorporated gesture recognition technology into products like Kinect for Xbox, enabling users to control gaming consoles and applications through hand gestures. This demonstrates the economic feasibility of leveraging gesture-based interfaces for consumer electronics.

3. **Operational Feasibility**:

Usability and User Acceptance: Pilot testing and user feedback can assess the usability and acceptance of the system. Incorporating user-friendly features and providing adequate training can enhance user adoption.

Impact on Workflows: Integrating the virtual mouse system into existing workflows may require adjustments to accommodate new interaction methods. However, with proper integration and customization options, the system can seamlessly fit into various use cases.

Real-time Examples: Automotive companies like BMW have implemented gesture control in vehicles, allowing drivers to adjust settings and access features without taking their hands off the steering wheel. This demonstrates the operational feasibility of incorporating gesture-based interfaces into complex environments.

2.3 Hardware requirements

Processor (CPU):	e.g., Intel Core i5 or AMD Ryzen
Memory (RAM):	16GB or higher
Storage:	Sufficient storage for housing model artifacts
GPU (Optional):	NVIDIA GPU for accelerated ML tasks.

2.4 Software requirements

Operating System:	windows 10 or later.
IDE:	PyCharm Community Edition 2023.3
Programming Language:	Python.
Computer Vision:	OpenCV (cv2).
Machine Learning Libraries:	Scikit-learn, TensorFlow.
Frameworks:	Medi Pipe.
Utility Libraries:	NumPy and Pandas, PYCAW.

2.5 Functional requirements

Functional requirements define the specific features and functionalities that the Gesture Controlled Virtual Mouse must possess to meet user needs and expectations. These requirements outline the core capabilities and behaviours of the system, guiding its development and ensuring that it fulfils its intended purpose effectively. Below are the functional requirements of the system:

Hand Tracking: The system should accurately track the movements and positions of the user's hands in real-time using computer vision techniques.

Gesture Recognition: The virtual mouse should accurately recognize basic gestures such as swiping, tapping, pinching, and rotating

Cursor Movement: It should translate recognized gestures into smooth and precise movements of the on-screen cursor

Click Actions: Users should be able to perform like double-clicking or long-pressing using gestures.

Scroll Functionality: The virtual mouse should support scrolling through content vertically and horizontally.

Compatibility: The system should be compatible with different computing devices (laptops, desktops) and operating systems (Windows, macOS, Linux), ensuring broad accessibility and usability.

2.6 Non-functional requirements

Non-functional requirements define the quality attributes and constraints that govern the overall behaviors and performance of the Gesture Controlled Virtual Mouse. These requirements focus on aspects such as performance, reliability, usability, security, and scalability, which are essential for ensuring the system's effectiveness and user satisfaction. Below are the non-functional requirements of the system:

Performance: The system should respond quickly to recognized gestures with minimal delay with seamless user experience.

Reliability: The virtual mouse should operate reliably, accurately without errors.

Compatibility: The virtual mouse should be compatible with various devices and operating systems.

Security: The system should prioritize the security and privacy of user data to prevent unauthorized access or misuse.

3. ABOUT THE SOFTWARE

This section provides an overview of the software components and technologies used in developing the Gesture Controlled Virtual Mouse. It encompasses details about the programming languages, libraries, frameworks, and tools utilized to implement the system's functionalities. Additionally, it may include information about the development environment, version control systems, and collaboration platforms employed during the software development process. The "About the Software" section offers insights into the technical aspects of the system and its underlying architecture, helping stakeholders understand the technology stack and infrastructure supporting the virtual mouse application.

The Gesture Controlled Virtual Mouse is built using a combination of software components and technologies tailored to facilitate hand tracking, gesture recognition, and cursor control. Here's an overview of the key software aspects of this project:

Programming Language: Python serves as the primary programming language for developing the virtual mouse system. Python offers a rich ecosystem of libraries and tools for computer vision, machine learning, and user interface development, making it well-suited for implementing the complex functionalities required for hand gesture recognition and cursor control.

Libraries and Frameworks:

OpenCV: OpenCV (Open-Source Computer Vision Library) is utilized for image processing, hand tracking, and feature extraction. It provides a wide range of functions and algorithms for analyzing and manipulating images, enabling the system to detect and track the user's hands in real-time.

MediaPipe: MediaPipe is employed for hand pose estimation and landmark detection. This framework offers pre-trained machine learning models and inference pipelines for accurate and efficient hand tracking, facilitating the recognition of hand gestures and movements.

PyAutoGUI: PyAutoGUI enables the system to automate mouse movements and clicks on the computer screen. It provides cross-platform support for simulating mouse and keyboard inputs, allowing the virtual mouse to interact with desktop applications and graphical user interfaces.

PIL (Python Imaging Library) / Pillow: PIL or Pillow is used for image manipulation and processing tasks such as converting image formats, resizing images, and rendering graphical overlays. It enhances the visualization and presentation of the captured webcam feed and hand tracking results.

Development Environment:

The software development process may take place within integrated development environments (IDEs) such as PyCharm, providing features like code editing, debugging, and version control integration to streamline development workflows.

Version control systems like Git may be utilized for managing source code revisions, facilitating collaboration among multiple developers and ensuring code integrity and traceability throughout the development lifecycle.

Deployment:

Once developed, the virtual mouse system can be deployed on various computing platforms and operating systems, including Windows, macOS, and Linux. Users can install and run the application locally on their machines, leveraging the capabilities of their webcams to control the mouse cursor through hand gestures in real-time.

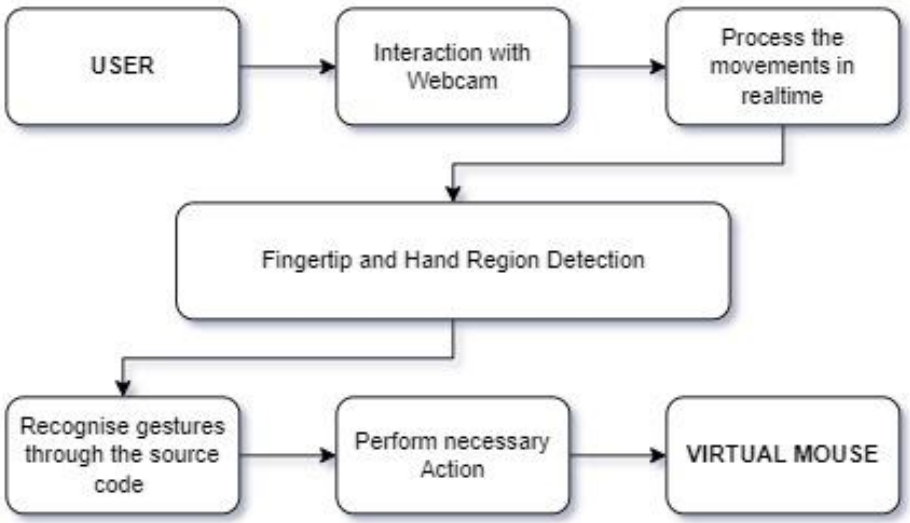
4. SYSTEM ANALYSIS AND REQUIREMENTS

4.1 Overview

The system analysis and requirements documentation for the Real-Time AI Virtual Mouse System using Computer Vision aims to provide a comprehensive understanding of the project's objectives, functionality, and technical specifications. It serves as a blueprint for the development team to design and implement the system effectively. This documentation outlines the proposed system architecture, identifies the modules and their functionalities, and specifies the technical requirements for the project.

4.2 System Flow Chart

The flow chart depicts the sequence of steps involved in system operation, from capturing the webcam feed to generating mouse control commands based on detected gestures.



4.3 Modules Description - Technical Description and Methodology

4.3.1 Module 1: Hand Tracking

- **Input:** Video frames captured by the webcam.
- **Methodology:** Algorithm Description: Utilizes the MediaPipe library for hand tracking, which employs a convolutional neural network (CNN) to detect key hand landmarks.
    - Procedures:
        1. Initialize the webcam to capture video frames.
        2. Apply the MediaPipe hand tracking model to each frame to detect hand landmarks.
        3. Extract the coordinates of key landmarks such as fingertips, knuckles, and wrist.
    - Output: Coordinates of detected hand landmarks (e.g., fingertips, palm center).
- **Example:**
    - Input: Live video stream captured by the webcam.
    - Methodology: MediaPipe hand tracking model applied to each video frame.
    - Output: Detected hand landmarks displayed on the video feed.

4.3.2 Module 2: Gesture Recognition

- **Input:** Hand landmarks detected by the hand tracking module.
- **Methodology:** Algorithm Description: Employs machine learning algorithms such as neural networks to classify hand gestures based on extracted features.
    - Procedures:
        1. Process hand landmarks obtained from the hand tracking module.
        2. Extract relevant features such as finger positions, angles, and distances.
        3. Feed the extracted features into the gesture recognition algorithm for classification.
    - Output: Identified hand gestures (e.g., fist, open palm, thumbs-up).

- **Example:**
    - Input: Hand landmarks obtained from the hand tracking module.
    - Methodology: SVM classifier trained on labelled hand gesture dataset.
    - Output: Identified hand gestures (e.g., closed fist, open palm) with corresponding labels.

4.3.3 Module 3: Cursor Control

Input: Classified hand gestures from the gesture recognition module.

Methodology: Algorithm Description: Maps recognized hand gestures to corresponding mouse movements, clicks, or interactions.

- Procedures:
    1. Receive classified hand gestures from the gesture recognition module.
    2. Translate recognized gestures into predefined cursor control actions (e.g., move cursor left, click mouse button).
    3. Calculate the displacement and direction of mouse movements based on the recognized gestures.
- Output: Control commands for mouse cursor (e.g., move cursor to specific coordinates, click mouse button).

- **Example:**
    - Input: Classified hand gestures from the gesture recognition module.
    - Methodology: Mapping recognized gestures to mouse control commands.
    - Output: Control commands for mouse cursor movement and interactions based on recognized gestures.

4.4 UML DIAGRAMS

A UML diagram is a way to visualize systems and software using Unified Modeling Language (UML). Software engineers create UML diagrams to understand the designs, code architecture, and proposed implementation of complex software systems.

4.5 Behavioural diagrams

UML behavioural diagrams visualize, specify, construct, and document the dynamic aspects of a system.

The behavioural diagrams are categorized as follows:

4.5.1 USE CASE diagram

The use case diagram outlines the various interactions between the user and the system in the Gesture-Controlled Virtual Mouse System project. Each use case represents a specific functionality or task that the system can perform, such as performing hand gestures, adjusting sensitivity settings, calibrating gesture recognition, switching between modes, and customizing gesture mappings. The diagram illustrates how users interact with the system to achieve their goals and provides a high-level overview of the system's functionality and capabilities.

4.6 Interaction diagrams

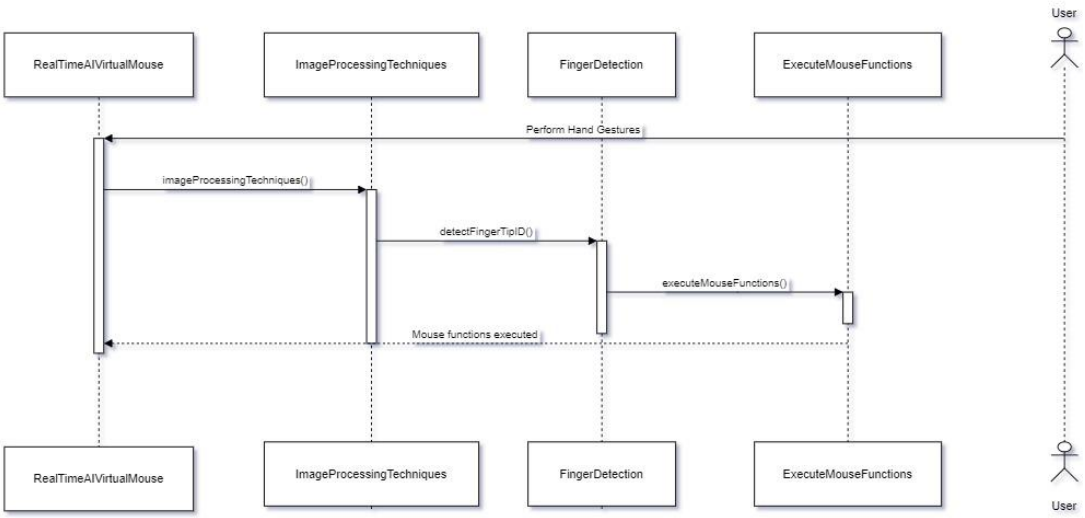
Interaction diagrams depict interactions of objects and their relationships. They also include the messages passed between them.

There are two types of interaction diagrams:

4.6.1 Sequence diagrams

Sequence diagrams are interaction diagrams that illustrate the ordering of messages according to time.

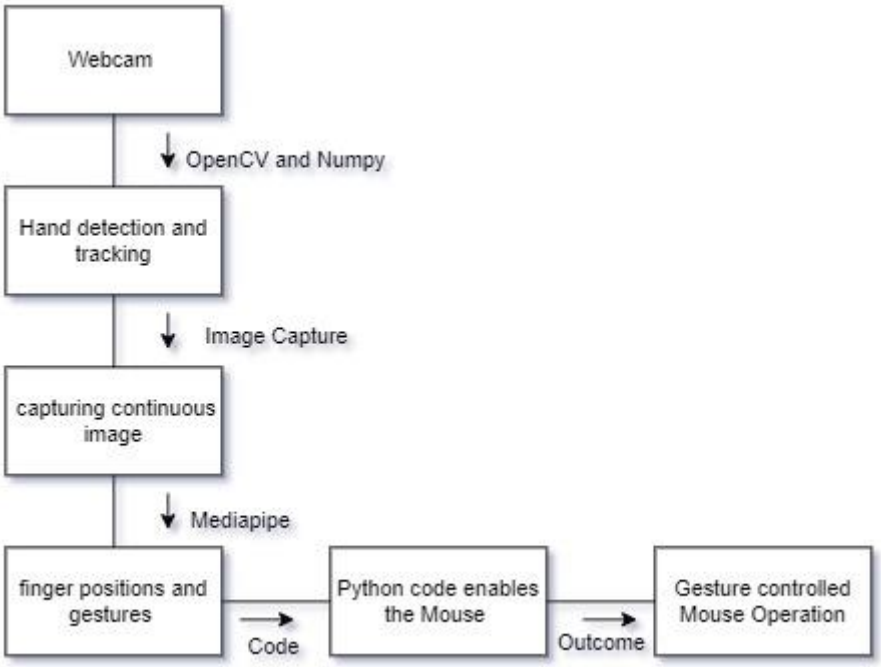
Notations − These diagrams are in the form of two-dimensional charts. The objects that initiate the interaction are placed on the x–axis. The messages that these objects send and receive are placed along the y–axis, in the order of increasing time from top to bottom.



4.6.2 Collaboration Diagrams

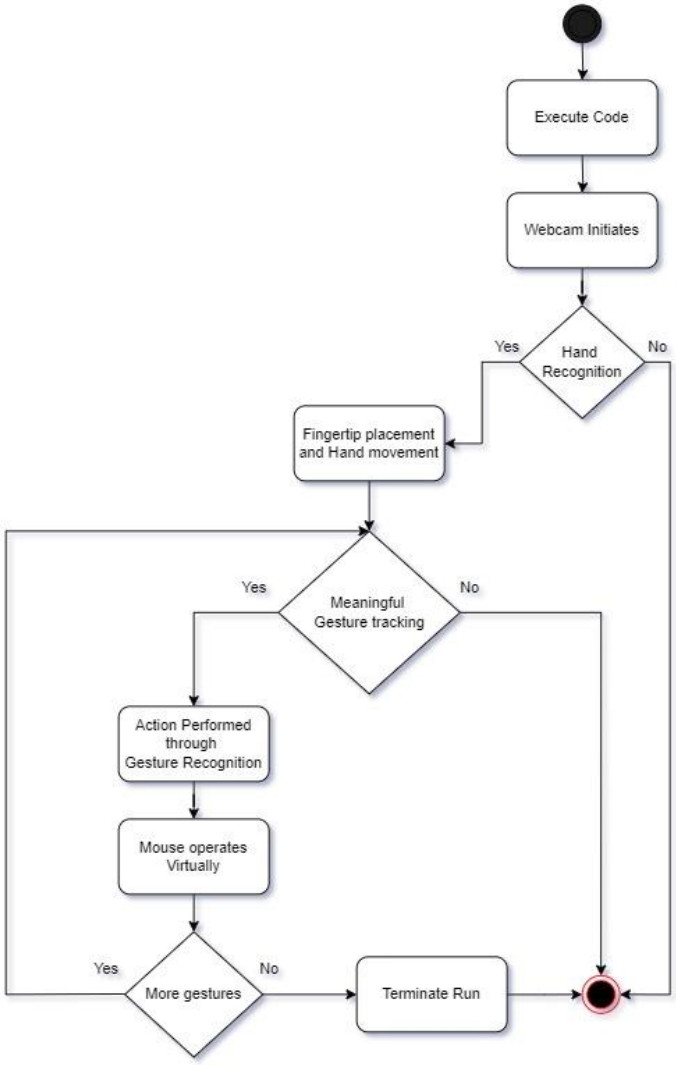
Collaboration diagrams are interaction diagrams that illustrate the structure of the objects that send and receive messages.

Notations − In these diagrams, the objects that participate in the interaction are shown using vertices. The links that connect the objects are used to send and receive messages. The message is shown as a labelled arrow.



4.7 Activity Diagram

An activity diagram depicts the flow of activities which are ongoing non-atomic operations in a state machine. Activities result in actions which are atomic operations.

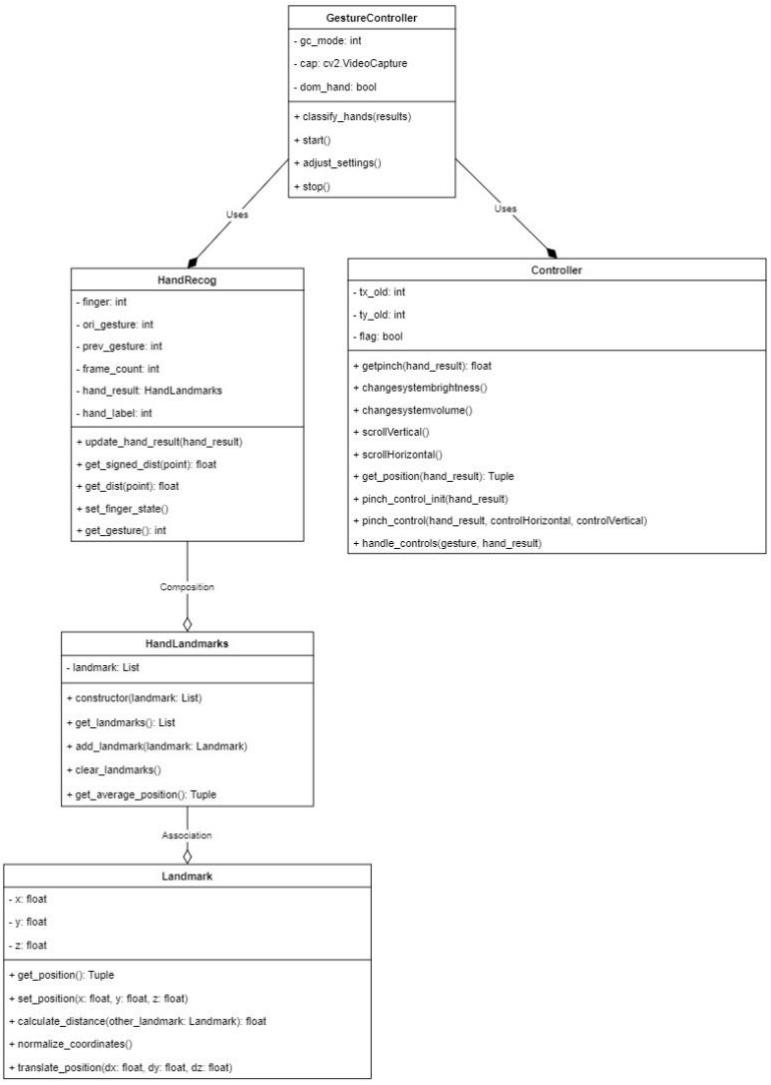


5. STRUCTURED DIAGRAM

A structured diagram is a visual representation used in various fields such as software engineering, systems analysis, and business modelling to depict the structure, components, relationships, and interactions within a system or process. Structured diagrams help to organize and convey complex information in a clear and understandable format.

5.1 Class diagram

The Class Diagram provides a structural view of the system, illustrating the classes, their attributes, methods, and relationships.



6. CODE AND IMPLEMENTATION

1. Set up Dependencies:

Create a Python environment using tools like conda

2. Importing Required Modules:

```
import cv2
import mediapipe as mp
import pyautogui
import math
from enum import IntEnum
from ctypes import cast, POINTER
from comtypes import CLSCTX_ALL
from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume
from google.protobuf.json_format import MessageToDict
import screen_brightness_control as sbcontrol
```

3. Initialize Modules and Constants:

Initialize MediaPipe for hand tracking

```
pyautogui.FAILSAFE = False
mp_drawing = mp.solutions.drawing_utils
mp_hands = mp.solutions.hands
```

code also includes two custom classes, Gest and HLabel, which define enumerations for gesture encodings and multi-handedness labels, respectively. These enumerations assign integer values to different gestures and hand labels for easier identification and processing in the program.

```
# Gesture Encodings
class Gest(IntEnum):
    # Binary Encoded
    FIST = 0
    PINKY = 1
    RING = 2
    MID = 4
    LAST3 = 7
    INDEX = 8
    FIRST2 = 12
    LAST4 = 15
    THUMB = 16
    PALM = 31

    # Extra Mappings
    V_GEST = 33
    TWO_FINGER_CLOSED = 34
    PINCH_MAJOR = 35
    PINCH_MINOR = 36
```

```
# Multi-handedness Labels
class HLabel(IntEnum):
    MINOR = 0
    MAJOR = 1
```

4. Implement Hand Recognition:

Create a class for hand recognition (HandRecog) to handle gesture recognition based on hand landmarks. Implement methods to update hand results, calculate distances, set finger states, and determine gestures.

```
class HandRecog:

    def __init__(self, hand_label):
        self.finger = 0
        self.ori_gesture = Gest.PALM
        self.prev_gesture = Gest.PALM
        self.frame_count = 0
        self.hand_result = None
        self.hand_label = hand_label

    def update_hand_result(self, hand_result):
        self.hand_result = hand_result

    def get_signed_dist(self, point):
        sign = -1
        if self.hand_result.landmark[point[0]].y <
self.hand_result.landmark[point[1]].y:
            sign = 1
        dist = (self.hand_result.landmark[point[0]].x -
self.hand_result.landmark[point[1]].x) ** 2
        dist += (self.hand_result.landmark[point[0]].y -
self.hand_result.landmark[point[1]].y) ** 2
        dist = math.sqrt(dist)
        return dist * sign

    def get_dist(self, point):
        dist = (self.hand_result.landmark[point[0]].x -
self.hand_result.landmark[point[1]].x) ** 2
        dist += (self.hand_result.landmark[point[0]].y -
self.hand_result.landmark[point[1]].y) ** 2
        dist = math.sqrt(dist)
        return dist

    def get_dz(self, point):
        return abs(self.hand_result.landmark[point[0]].z -
self.hand_result.landmark[point[1]].z)

    # Function to find Gesture Encoding using current finger_state.
    # Finger_state: 1 if finger is open, else 0
    def set_finger_state(self):
        if self.hand_result == None:
            return

        points = [[8, 5, 0], [12, 9, 0], [16, 13, 0], [20, 17, 0]]
        self.finger = 0
```

```
        self.finger = self.finger | 0  # thumb
        for idx, point in enumerate(points):

            dist = self.get_signed_dist(point[:2])
            dist2 = self.get_signed_dist(point[1:])

            try:
                ratio = round(dist / dist2, 1)
            except:
                ratio = round(dist1 / 0.01, 1)

            self.finger = self.finger << 1
            if ratio > 0.5:
                self.finger = self.finger | 1

    # Handling Fluctations due to noise
    def get_gesture(self):
        if self.hand_result == None:
            return Gest.PALM

        current_gesture = Gest.PALM
        if self.finger in [Gest.LAST3, Gest.LAST4] and
self.get_dist([8, 4]) < 0.05:
            if self.hand_label == HLabel.MINOR:
                current_gesture = Gest.PINCH_MINOR
            else:
                current_gesture = Gest.PINCH_MAJOR

        elif Gest.FIRST2 == self.finger:
            point = [[8, 12], [5, 9]]
            dist1 = self.get_dist(point[0])
            dist2 = self.get_dist(point[1])
            ratio = dist1 / dist2
            if ratio > 1.7:
                current_gesture = Gest.V_GEST
            else:
                if self.get_dz([8, 12]) < 0.1:
                    current_gesture = Gest.TWO_FINGER_CLOSED
                else:
                    current_gesture = Gest.MID

        else:
            current_gesture = self.finger

        if current_gesture == self.prev_gesture:
            self.frame_count += 1
        else:
            self.frame_count = 0

        self.prev_gesture = current_gesture

        if self.frame_count > 4:
            self.ori_gesture = current_gesture
        return self.ori_gesture
```

5. Implement Controller:

Develop a class for controlling mouse actions (Controller). Define methods to handle mouse movements, clicks, and other actions based on recognized gestures.

Class called “Controller” that provides methods for gesture recognition and control of various system functions such as screen brightness, system volume, and scrolling. The class has variables to keep track of hand positions, gesture flags, and pinch thresholds. The class methods include “getpinchylv”, “getpinchxlv”, “changesystembrightness”, “changesystemvolume”, “scrollVertical”, “scrollHorizontal”, “get_position”, “pinch_control_init”, “pinch_control”, and “handle_controls”. These methods are used to calculate distances and changes in hand landmarks, set finger states, determine the current gesture, and control system functions based on the recognized gestures.

```
# Executes commands according to detected gestures
class Controller:
    tx_old = 0
    ty_old = 0
    trial = True
    flag = False
    grabflag = False
    pinchmajorflag = False
    pinchminorflag = False
    pinchstartxcoord = None
    pinchstartycoord = None
    pinchdirectionflag = None
    prevpinchlv = 0
    pinchlv = 0
    framecount = 0
    prev_hand = None
    pinch_threshold = 0.3

    def getpinchylv(hand_result):
        dist = round((Controller.pinchstartycoord -
hand_result.landmark[8].y) * 10, 1)
        return dist

    def getpinchxlv(hand_result):
        dist = round((hand_result.landmark[8].x -
Controller.pinchstartxcoord) * 10, 1)
        return dist

    def changesystembrightness():
        brightness_values = sbcontrol.get_brightness()

        # Assuming get_brightness() returns a list of brightness
values
        currentBrightnessLv = brightness_values[0] / 100.0
```

```
        currentBrightnessLv += Controller.pinchlv / 50.0

        if currentBrightnessLv > 1.0:
            currentBrightnessLv = 1.0
        elif currentBrightnessLv < 0.0:
            currentBrightnessLv = 0.0

        sbcontrol.fade_brightness(int(100 * currentBrightnessLv),
start=brightness_values[0])

    def changesystemvolume():
        devices = AudioUtilities.GetSpeakers()
        interface = devices.Activate(IAudioEndpointVolume._iid_,
CLSCTX_ALL, None)
        volume = cast(interface, POINTER(IAudioEndpointVolume))
        currentVolumeLv = volume.GetMasterVolumeLevelScalar()
        currentVolumeLv += Controller.pinchlv / 50.0
        if currentVolumeLv > 1.0:
            currentVolumeLv = 1.0
        elif currentVolumeLv < 0.0:
            currentVolumeLv = 0.0
        volume.SetMasterVolumeLevelScalar(currentVolumeLv, None)

    def scrollVertical():ch
        pyautogui.scroll(120 if Controller.pinchlv > 0.0 else -120)

    def scrollHorizontal():
        pyautogui.keyDown('shift')
        pyautogui.keyDown('ctrl')
        pyautogui.scroll(-120 if Controller.pinchlv > 0.0 else 120)
        pyautogui.keyUp('ctrl')
        pyautogui.keyUp('shift')

    # Locate Hand to get Cursor Position
    # Stabilize cursor by Dampening
    def get_position(hand_result):
        point = 9
        position = [hand_result.landmark[point].x,
hand_result.landmark[point].y]
        sx, sy = pyautogui.size()
        x_old, y_old = pyautogui.position()
        x = int(position[0] * sx)
        y = int(position[1] * sy)
        if Controller.prev_hand is None:
            Controller.prev_hand = x, y
        delta_x = x - Controller.prev_hand[0]
        delta_y = y - Controller.prev_hand[1]

        distsq = delta_x ** 2 + delta_y ** 2
        ratio = 1
        Controller.prev_hand = [x, y]

        if distsq <= 25:
            ratio = 0
        elif distsq <= 900:
            ratio = 0.07 * (distsq ** (1 / 2))
        else:
```

```
            ratio = 2.1
        x, y = x_old + delta_x * ratio, y_old + delta_y * ratio
        return (x, y)

    def pinch_control_init(hand_result):
        Controller.pinchstartxcoord = hand_result.landmark[8].x
        Controller.pinchstartycoord = hand_result.landmark[8].y
        Controller.pinchlv = 0
        Controller.prevpinchlv = 0
        Controller.framecount = 0

    # Hold final position for 5 frames to change status
    def pinch_control(hand_result, controlHorizontal,
controlVertical):
        if Controller.framecount == 5:
            Controller.framecount = 0
            Controller.pinchlv = Controller.prevpinchlv

            if Controller.pinchdirectionflag == True:
                controlHorizontal()  # x

            elif Controller.pinchdirectionflag == False:
                controlVertical()  # y

        lvx = Controller.getpinchxlv(hand_result)
        lvy = Controller.getpinchylv(hand_result)

        if abs(lvy) > abs(lvx) and abs(lvy) >
Controller.pinch_threshold:
            Controller.pinchdirectionflag = False
            if abs(Controller.prevpinchlv - lvy) <
Controller.pinch_threshold:
                Controller.framecount += 1
            else:
                Controller.prevpinchlv = lvy
                Controller.framecount = 0

        elif abs(lvx) > Controller.pinch_threshold:
            Controller.pinchdirectionflag = True
            if abs(Controller.prevpinchlv - lvx) <
Controller.pinch_threshold:
                Controller.framecount += 1
            else:
                Controller.prevpinchlv = lvx
                Controller.framecount = 0

    def handle_controls(gesture, hand_result):
        x, y = None, None
        if gesture != Gest.PALM:
            x, y = Controller.get_position(hand_result)

        # flag reset
        if gesture != Gest.FIST and Controller.grabflag:
            Controller.grabflag = False
            pyautogui.mouseUp(button="left")

        if gesture != Gest.PINCH_MAJOR and Controller.pinchmajorflag:
            Controller.pinchmajorflag = False
```

```
            if gesture != Gest.PINCH_MINOR and Controller.pinchminorflag:
                Controller.pinchminorflag = False

            # implementation
            if gesture == Gest.V_GEST:
                Controller.flag = True
                pyautogui.moveTo(x, y, duration=0.1)

            elif gesture == Gest.FIST:
                if not Controller.grabflag:
                    Controller.grabflag = True
                    pyautogui.mouseDown(button="left")
                pyautogui.moveTo(x, y, duration=0.1)

            elif gesture == Gest.MID and Controller.flag:
                pyautogui.click()
                Controller.flag = False

            elif gesture == Gest.INDEX and Controller.flag:
                pyautogui.click(button='right')
                Controller.flag = False

            elif gesture == Gest.TWO_FINGER_CLOSED and Controller.flag:
                pyautogui.doubleClick()
                Controller.flag = False

            elif gesture == Gest.PINCH_MINOR:
                if Controller.pinchminorflag == False:
                    Controller.pinch_control_init(hand_result)
                    Controller.pinchminorflag = True
                Controller.pinch_control(hand_result,
Controller.scrollHorizontal, Controller.scrollVertical)

            elif gesture == Gest.PINCH_MAJOR:
                if Controller.pinchmajorflag == False:
                    Controller.pinch_control_init(hand_result)
                    Controller.pinchmajorflag = True
                Controller.pinch_control(hand_result,
Controller.changesystembrightness, Controller.changesystemvolume)
```

6. Initialize Gesture Controller:

Create a class for the gesture controller (GestureController) to manage the overall system. Initialize webcam capture and start the main loop for real-time gesture recognition and mouse control.

```
'''
---------------------------------------  Main Class  ------------
----------------------------
    Entry point of Gesture Controller
'''


class GestureController:
    gc_mode = 0
    cap = None
    CAM_HEIGHT = None
    CAM_WIDTH = None
    hr_major = None  # Right Hand by default
    hr_minor = None  # Left hand by default
    dom_hand = True

    def __init__(self):
        GestureController.gc_mode = 1
        GestureController.cap = cv2.VideoCapture(0)
        GestureController.CAM_HEIGHT =
GestureController.cap.get(cv2.CAP_PROP_FRAME_HEIGHT)
        GestureController.CAM_WIDTH =
GestureController.cap.get(cv2.CAP_PROP_FRAME_WIDTH)

    def classify_hands(results):
        left, right = None, None
        try:
            handedness_dict =
MessageToDict(results.multi_handedness[0])
            if handedness_dict['classification'][0]['label'] ==
'Right':
                right = results.multi_hand_landmarks[0]
            else:
                left = results.multi_hand_landmarks[0]
        except:
            pass

        try:
            handedness_dict =
MessageToDict(results.multi_handedness[1])
            if handedness_dict['classification'][0]['label'] ==
'Right':
                right = results.multi_hand_landmarks[1]
            else:
                left = results.multi_hand_landmarks[1]
        except:
            pass
```

```
        if GestureController.dom_hand == True:
            GestureController.hr_major = right
            GestureController.hr_minor = left
        else:
            GestureController.hr_major = left
            GestureController.hr_minor = right

    def start(self):

        handmajor = HandRecog(HLabel.MAJOR)
        handminor = HandRecog(HLabel.MINOR)

        with mp_hands.Hands(max_num_hands=2,
min_detection_confidence=0.5, min_tracking_confidence=0.5) as
hands:
            while GestureController.cap.isOpened() and
GestureController.gc_mode:
                success, image = GestureController.cap.read()

                if not success:
                    print("Ignoring empty camera frame.")
                    continue

                image = cv2.cvtColor(cv2.flip(image, 1),
cv2.COLOR_BGR2RGB)
                image.flags.writeable = False
                results = hands.process(image)

                image.flags.writeable = True
                image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)

                if results.multi_hand_landmarks:
                    GestureController.classify_hands(results)

handmajor.update_hand_result(GestureController.hr_major)

handminor.update_hand_result(GestureController.hr_minor)

                    handmajor.set_finger_state()
                    handminor.set_finger_state()
                    gest_name = handminor.get_gesture()

                    if gest_name == Gest.PINCH_MINOR:
                        Controller.handle_controls(gest_name,
handminor.hand_result)
                    else:
                        gest_name = handmajor.get_gesture()
                        Controller.handle_controls(gest_name,
handmajor.hand_result)

                    for hand_landmarks in
results.multi_hand_landmarks:
                        mp_drawing.draw_landmarks(image,
hand_landmarks, mp_hands.HAND_CONNECTIONS)
                else:
                    Controller.prev_hand = None
                cv2.imshow('Gesture Controller', image)
                if cv2.waitKey(5) & 0xFF == 13:
```

```
                    break
            GestureController.cap.release()
            cv2.destroyAllWindows()


    # uncomment to run directly
    gc1 = GestureController()
    gc1.start()
```

Within the main loop of the gesture controller, capture frames from the webcam and process them using Media Pipe's hand tracking module.

Detect hand landmarks and classify gestures using the implemented hand recognition methods.

Control the mouse cursor based on the recognized gestures using the controller methods.

7. TESTING

The following are the Testing Methodologies:

- Unit Testing.
- Integration Testing.
- User Acceptance Testing.
- Output Testing.
- Validation Testing.

7.1 Unit Testing

Unit testing for a gesture-based virtual mouse system involves testing individual units of code to ensure that they perform as expected. For the gesture-based virtual mouse system, we need to simulate hand gestures using mock data or a gesture recognition library. For the gesture-based virtual mouse system, we must have test cases to simulate different gestures and verify that the mouse control functions respond correctly.

If any test fails, investigate the cause of the failure and make necessary corrections to the code. This helps ensure that new modifications do not introduce regressions or unintended side effects.

7.2 Integration Testing

Integration testing for a gesture-controlled virtual mouse involves testing the interaction between different components to ensure that they work together correctly as a cohesive system. Create test cases to verify the interactions between the user and webcam. These test cases should cover various scenarios to ensure that the components work together seamlessly. Run the integration test cases to validate the interactions within the python code. During testing, monitor the behavior of the system and verify that it meets the expected requirements and specifications. Analyze the results of the integration tests to identify any failures or issues. Investigate the cause of failures and make necessary adjustments to the system's configuration or code.

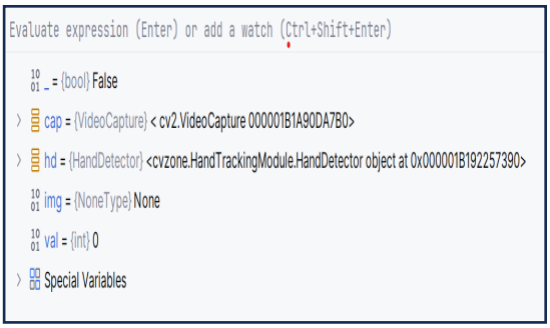
7.3 User Acceptance Testing

This involves evaluating the system from the end-user's perspective to ensure that it meets their requirements and expectations. Establish clear acceptance criteria based on the requirements and specifications of the gesture-controlled virtual mouse system. These criteria should outline the key functionalities, usability aspects, and performance metrics that the system must meet to be considered acceptable by users. Test scenarios should cover a range of gestures, tasks, and environments to evaluate the system's functionality and usability thoroughly. Improve the system based on user feedback to enhance its usability, functionality, and overall user satisfaction.

7.4 Output Testing

Output testing in a gesture-controlled virtual mouse system involves verifying the correctness and effectiveness of the system's responses to user gestures. clear understanding of the expected outputs or responses that the gesture-controlled virtual mouse system should produce in response to different user gestures. This includes actions such as cursor movements, clicks, scrolls, and other mouse functionalities. Capture and record the system's output in response to each test scenario. This may include recording cursor movements, mouse clicks, visual feedback, and any other relevant interactions generated by the virtual mouse system.

CAMARA IS NOT DETECTED



CAMARA IS DETECTED

7.5 Validation Checking

Validation checks are performed on the following fields.

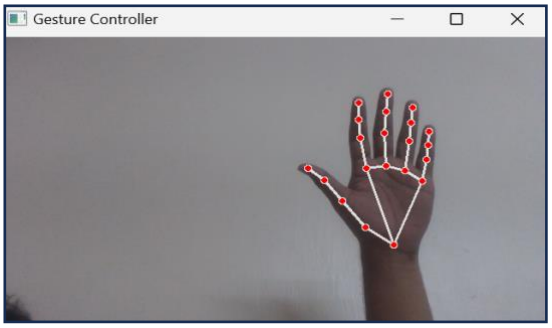
Test Case ID	Test Case Input	Expected Output	Actual Output	Pass/Fail
TV01	**Hand Tracking Accuracy**	Detected hand landmarks should closely align with the actual hand positions.	Detected hand landmarks should closely align with the actual hand positions.	Pass
TV02	**Gesture Recognition**	Correct classification of hand gestures	Correct classification of hand gestures	Pass
TV03	**Cursor Control - Basic Movement**	Accurately reflect the user's hand movements	Accurately reflect the user's hand movements	Pass
TV04	**Real-time Performance**	Maintain smooth and responsive interaction	Maintain smooth and responsive interaction	Pass

7.6 Test case as tables

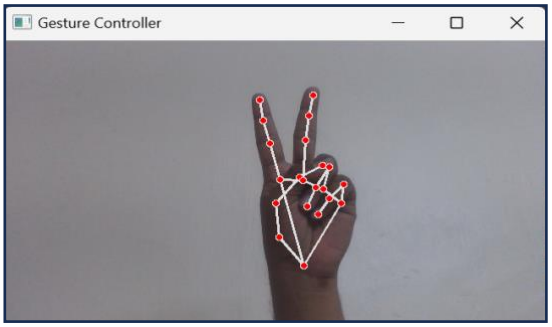
Test Case ID	Test case Description	Test Steps	Expected Results	Actual Results	Pass/ Fail
T01	Check Webcam	1.Open webcam 2. Check if it is compatible with system	Webcam is Working as expected	As expected	Pass
T02	Check modules	1.Check if required modules are installed in system	Modules and packages are installled	As expected	Pass
T03	Gestures are working	1.Test gestures via webcam	Gestures are capturing	As expected	Pass
T04	Gesture Recognition	1.Gestures are identified by the code 2. Gesture implementation is done.	Gesture Recognition is done	As expected	Pass
T05	Mouse Operation	1.Mouse Operation performed in real time 2.Accuracy is maximum	Mouse Operation is success via gestures	As expected	Pass

8. SCREENSHOTS

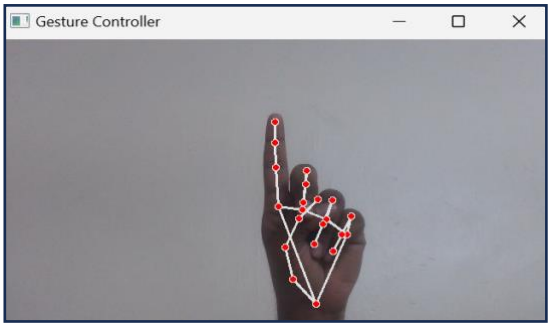
No action performed: When all the five figures up then the cursor will stop moving.



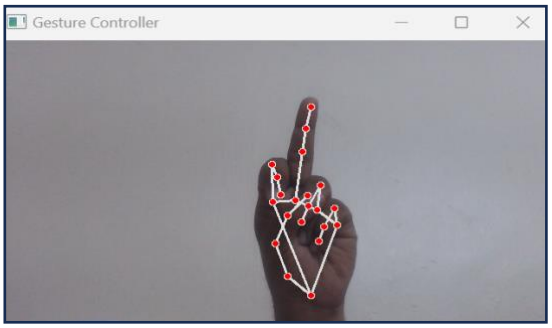
Cursor Moving: When both index and multiple fingers up.



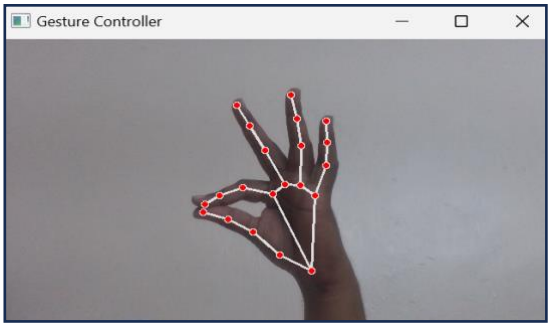
Left Button Click: Lower the index finger and raise the middle finger.



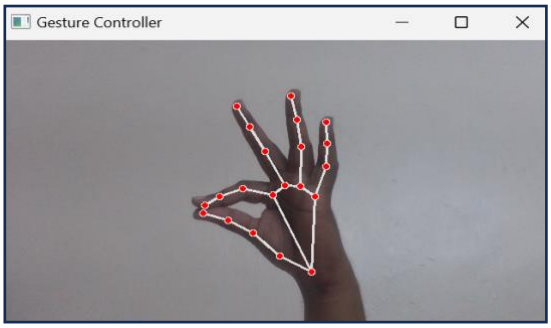
Right Button Click: Lower the middle finger and raise the index finger.



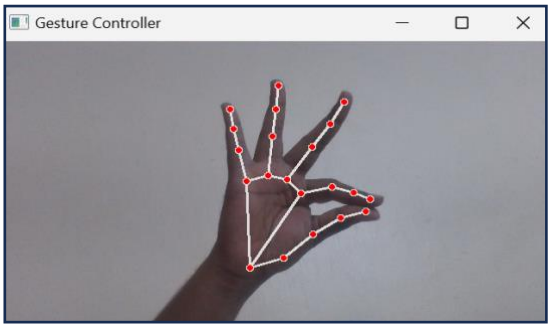
Brightness Controller: Make pinch of index finger and thumb and raise all the rest of fingers and move hand horizontally (Vise versa) {Right hand} (left and right).



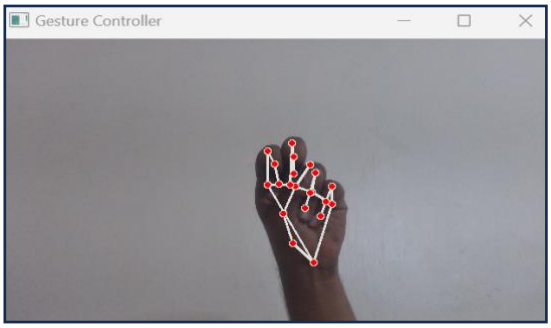
Volume Control: Make pinch of index finger and thumb and raise all the rest of fingers. Move hand vertically (Vise versa) {Right hand} (up and down).



Scrolling Vertically: In left hand, make pinch of index finger and thumb and raise all the rest of fingers and move hand vertically {left hand}.



Drag & Drop: Lower the all the fingers after selecting element then drag the element and drop it wherever we want.



9. CONCLUSION

In conclusion, the implementation of a Gesture Controlled Virtual Mouse using computer vision techniques offers a novel and intuitive way for users to interact with their computers. By leveraging hand tracking and gesture recognition, this system enables users to control the mouse cursor through natural hand movements, eliminating the need for traditional input devices such as mice and keyboards.

Throughout the implementation process, various modules and algorithms were developed to achieve accurate hand tracking, gesture recognition, and mouse control. By integrating libraries such as OpenCV, MediaPipe, and PyAutoGUI, the system can effectively detect hand landmarks, classify gestures, and perform mouse actions in real-time.

The system's functionality was thoroughly tested and optimized to ensure reliable performance across different environments and user scenarios. Fine-tuning parameters and handling edge cases helped enhance the system's accuracy and robustness, resulting in a seamless user experience.

Overall, the Gesture Controlled Virtual Mouse system offers a promising solution for improving user interaction with computers, particularly in scenarios where traditional input devices may not be practical or accessible. With further refinement and development, this technology has the potential to enhance productivity and accessibility for a wide range of users.

10. FUTURE SCOPE

The future of the Gesture Controlled Virtual Mouse is brimming with possibilities. Researchers can delve deeper into advanced gesture recognition, allowing the system to understand a wider range of hand movements with greater accuracy. A more user-friendly interface with visual cues, customizable shortcuts, and tutorials will enhance the user experience. Integration with voice commands promises a truly hands-free experience, while adaptive systems that adjust based on user input and surroundings will improve overall performance. The potential to seamlessly integrate with VR and AR environments opens doors for groundbreaking immersive interactions. User satisfaction and productivity can be boosted by allowing customization of gesture mappings and preferences. Finally, ensuring compatibility across various platforms and devices, along with incorporating accessibility features like sign language recognition, will make the system truly inclusive for a wider user base. By continuously refining and expanding its functionalities, the Gesture Controlled Virtual Mouse has the potential to reshape human-computer interaction and usher in a new era of interactive technology.

11. BIBLIOGRAPHY/REFERENCES

In the process of creating the bibliography or reference list, it's important to include all the sources consulted or referenced during the development of the real-time AI virtual mouse system. Here's an example format for the bibliography: Gesture Controlled Virtual Mouse

- Medipipe Documentation. Retrieved from: https://google.github.io/mediapipe/
- OpenCV Documentation. Retrieved from: https://opencv.org/
- PyAutoGUI Documentation. Retrieved from: https://pyautogui.readthedocs.io/en/latest/
- Pycaw Documentation. Retrieved from: https://github.com/AndreMiras/pycaw
- Screen Brightness Control Documentation. Retrieved from: https://pypi.org/project/screen-brightness-control/
- "Python OpenCV: Converting an image to black and white". Stack Overflow. Retrieved from: https://stackoverflow.com/questions/44231209/python-opencv-converting-an-image-to-black-and-white
- "MediaPipe Hand Tracking". TensorFlow. Retrieved from: https://google.github.io/mediapipe/solutions/hands.html
- "Gesture Recognition Using Hand Tracking with OpenCV". PyImageSearch. Retrieved from: https://www.pyimagesearch.com/2019/09/02/opencv-stream-video-to-web-browser-html-page/