AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department of Computer Engineering

Software Engineering Project Report

"Real-Time Hand Gesture Recognition and Control System using Python"

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AISSMS IOIT

S. Y. BTech Computer Engineering

SUBMITTED BY

Roll No.	Exam Seat Number	Name
89	SY2304082	Harsh R. Meshram

2023-24

AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department of Computer Engineering

CERTIFICATE

This is to certify that the project report

" Real-Time Hand Gesture Recognition and Control System using Python "

Submitted by

Student Roll No	Student Name
89	Harsh R. Meshram

is a bonafide student of this institute and the work has been carried out by her under the supervision of **Dr. Veena Bhende** and it is approved for the partial fulfillment of the Department of Computer Engineering AISSMS IOIT.

(Dr. Veena Bhende)
Guide
Department of Computer Engineering
Place : Pune

(Dr. S. N. ZAWARE)
Head,
Department of Computer Engineering
Date :

Index

CHAPTER 1: INTRODUCTION

1.1 TITLE

The title of my project is **"Real-Time Hand Gesture Recognition and Control System using Python"** enables intuitive computer interaction by tracking hand gestures and translating them into mouse movements and clicks.

1.2 OBJECTIVE

The objective of this project is to develop an advanced gesture recognition system that enables users to control their computer through intuitive hand gestures captured by a webcam. Utilizing cutting-edge computer vision techniques and machine learning models for hand tracking and gesture classification, the system aims to interpret specific hand movements and translate them into practical commands for system operations. This includes functionalities such as moving the mouse cursor, clicking, scrolling, adjusting volume, and controlling screen brightness. The project seeks to enhance user interaction by providing a hands-free, real-time, and user-friendly interface, thereby improving accessibility and offering an innovative method of computer control for various applications.

1.3 FEATURES OF THE PROJECT

The features of the "Real-Time Hand Gesture Recognition and Control System using Python" project are:

1. **Real-time Hand Gesture Recognition:** Utilizes AI algorithms to accurately detect and interpret hand gestures in real-time.

2. **Intuitive Cursor Control:** Enables users to control the mouse cursor using hand movements, providing a natural and intuitive interaction experience.

3. **Customizable Gestures:** Offers the flexibility for users to customize hand gestures for specific mouse actions, allowing for personalized control preferences.

4. **Adaptive Sensitivity:** Adjusts cursor movement sensitivity dynamically based on hand motion, ensuring precise control without being overly sensitive or unresponsive.

5. **Cross-Platform Compatibility:** Supports multiple operating systems such as Windows, macOS, and Linux, making it accessible to a wide range of users across different platforms.

6. **Accessibility Features:** Incorporates accessibility features to cater to users with disabilities, such as adjustable cursor speed and voice commands for mouse actions.

7. **Gesture Feedback:** Provides visual and/or auditory feedback to users upon successful recognition of hand gestures, enhancing the user experience and confirming input actions.

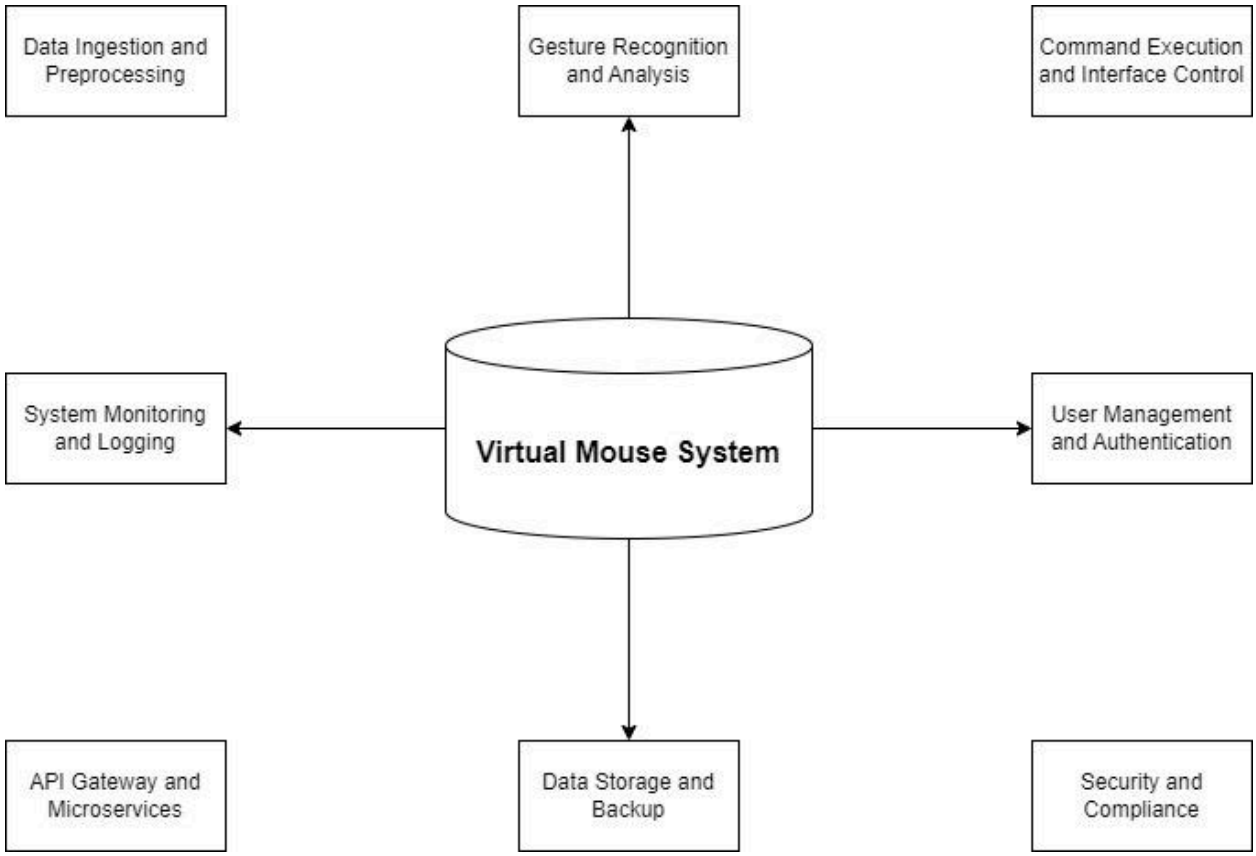
8. **Low Latency Interaction:** Minimizes latency between hand gestures and cursor movement, delivering a responsive and seamless interaction between the user and the virtual mouse.

9. **Robust Error Handling:** Implements error detection and recovery mechanisms to handle unexpected scenarios, ensuring the stability and reliability of the virtual mouse system.

10. **Open-Source Development:** Offers the project as open-source software, allowing for community contributions, feedback, and continuous improvement of the virtual mouse functionality.

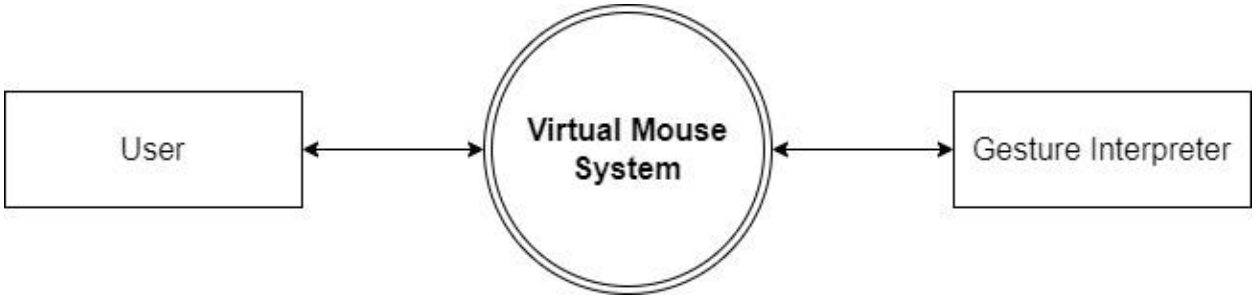
CHAPTER 2: DESIGN AND ARCHITECTURE

2.1 DATA CENTER ARCHITECTURE

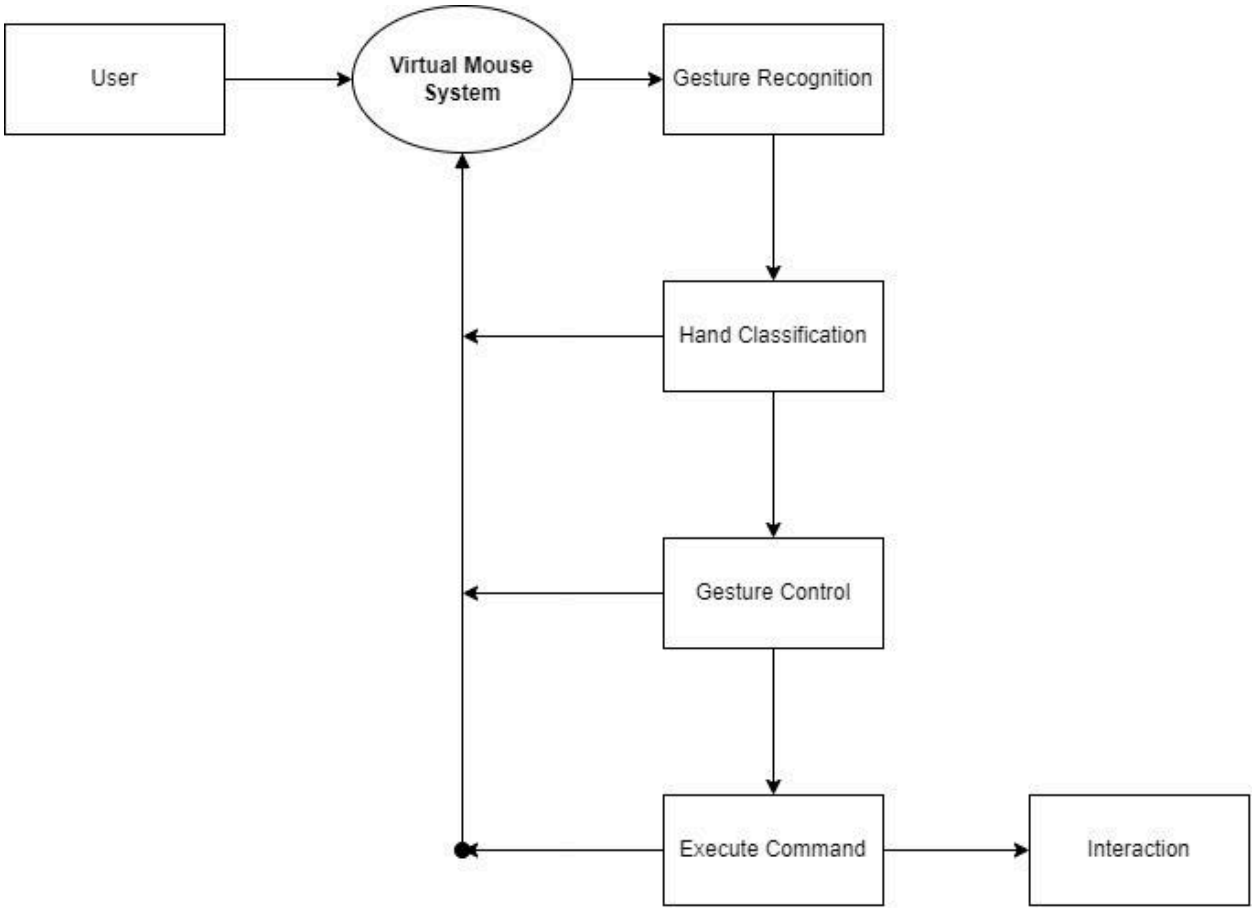


2.2 DATA FLOW ARCHITECTURE

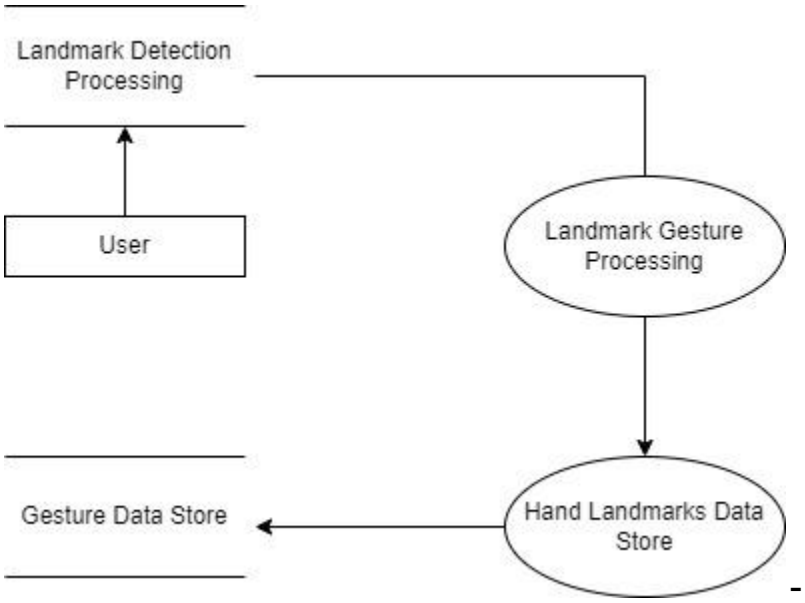
2.2.1 DFD LEVEL 0:



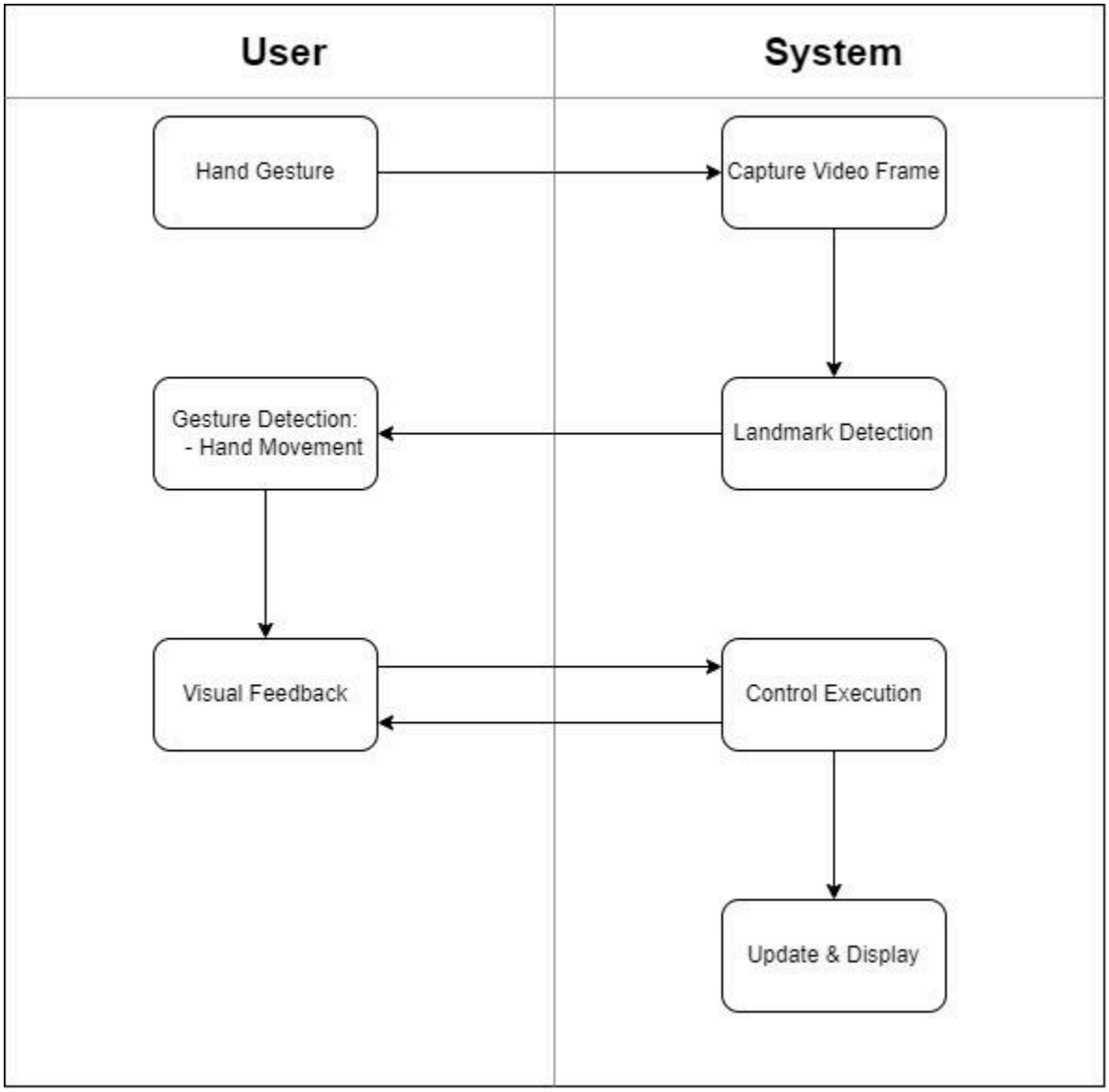
2.2.2 DFD LEVEL 1:



2.2.3 DFD LEVEL 2:

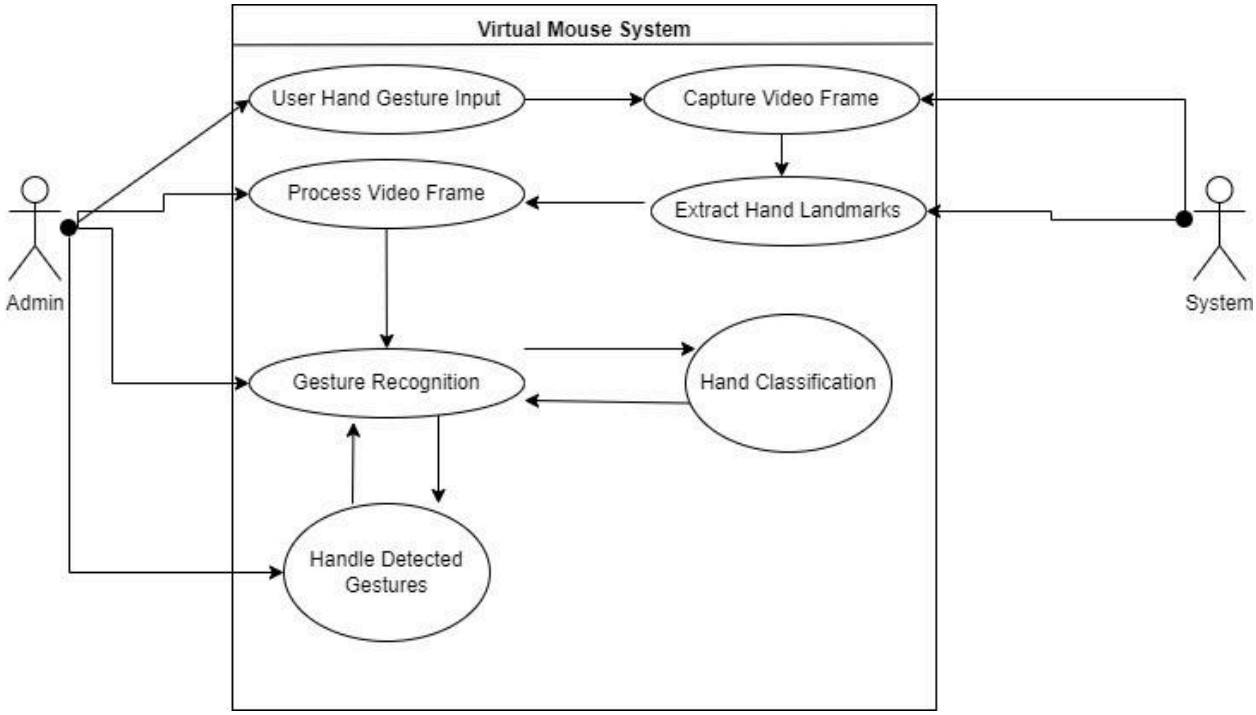


2.3 CALL AND RETURN ARCHITECTURE

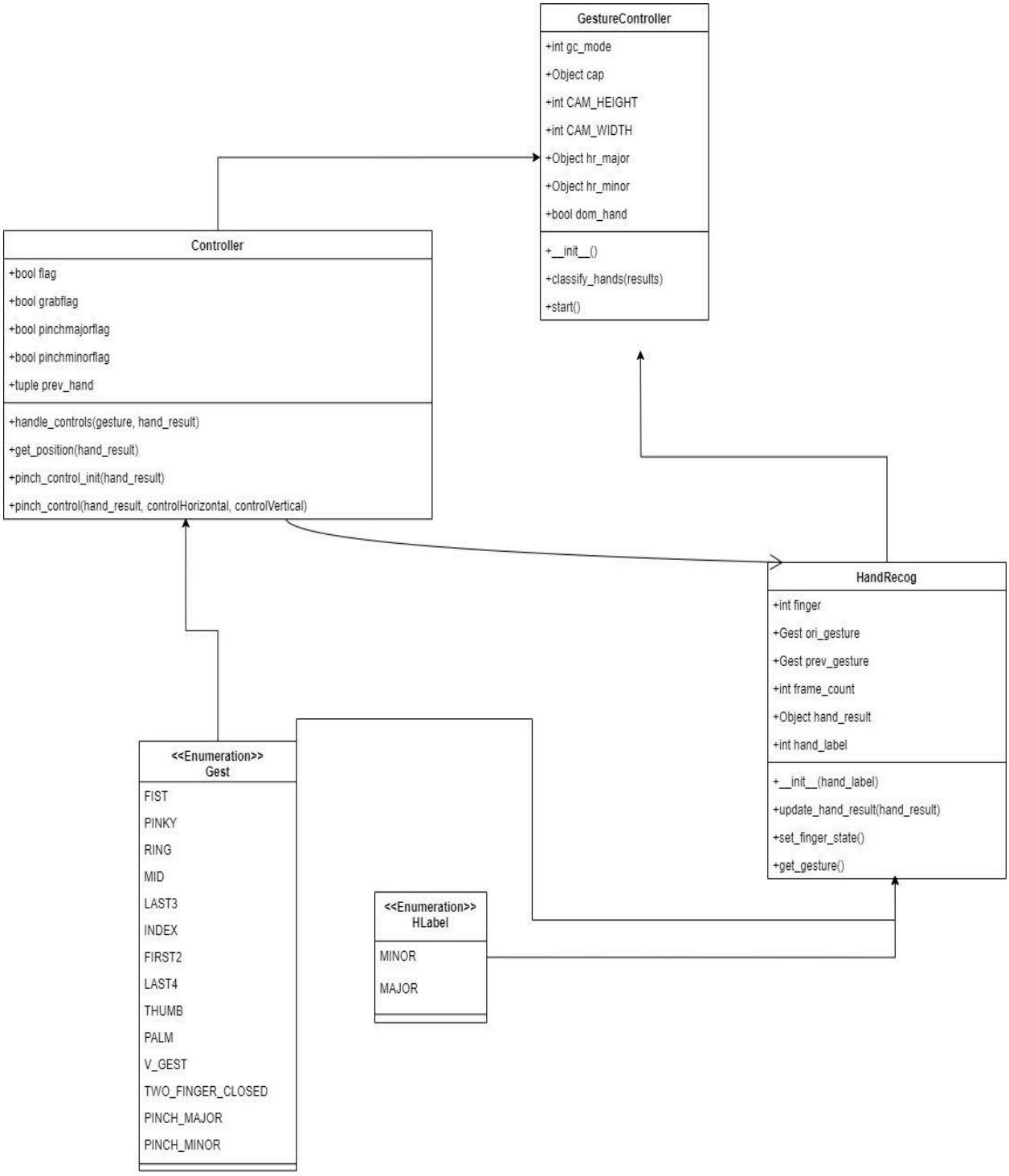


CHAPTER 3: UML DIAGRAM

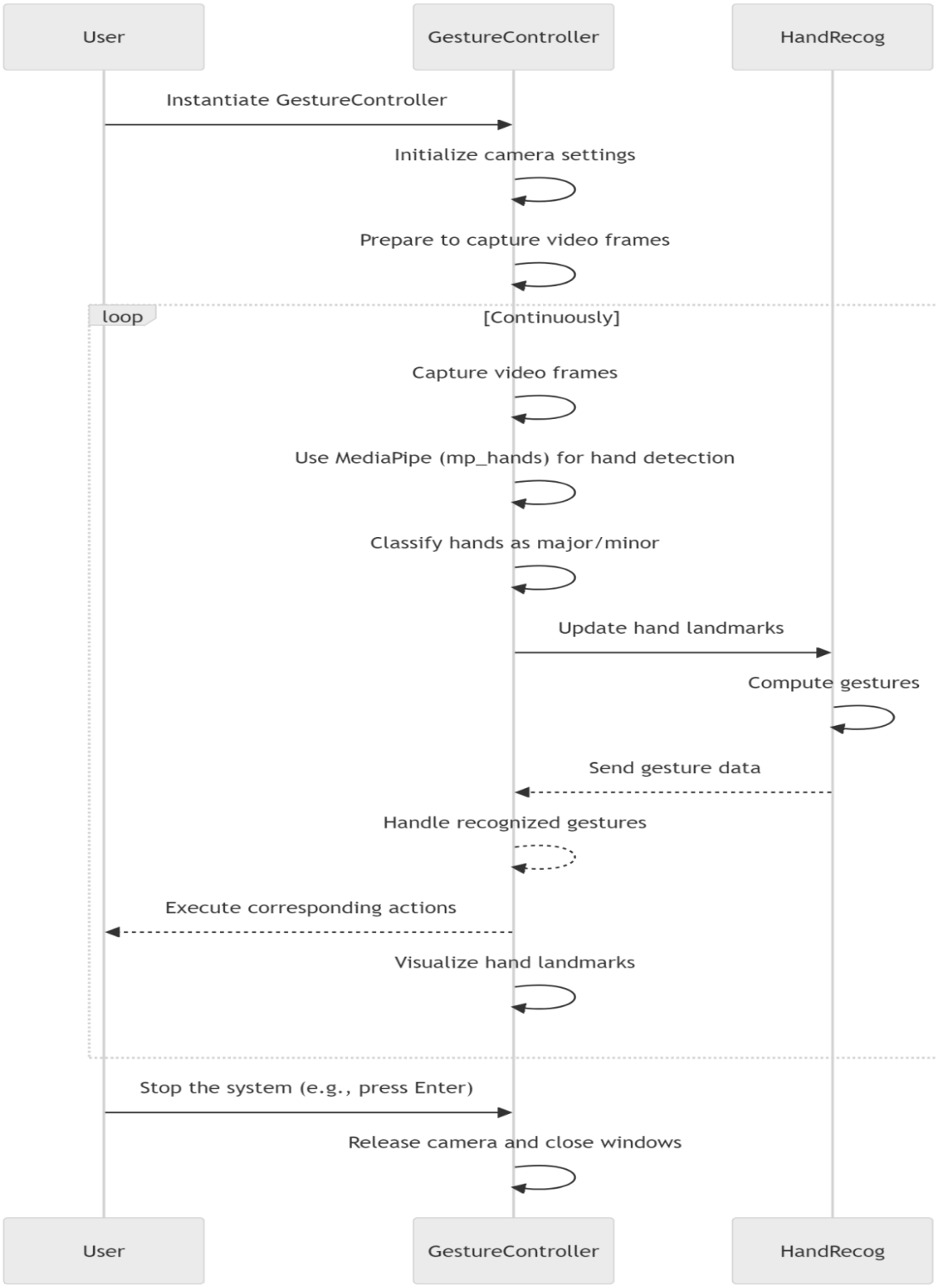
3.1 USE CASE DIAGRAM



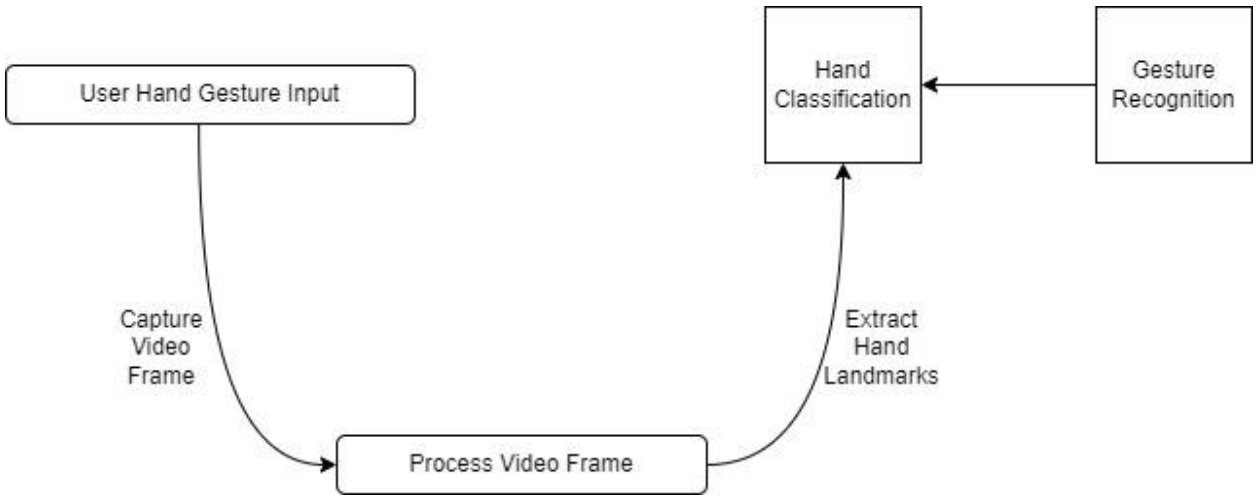
3.2 CLASS DIAGRAM



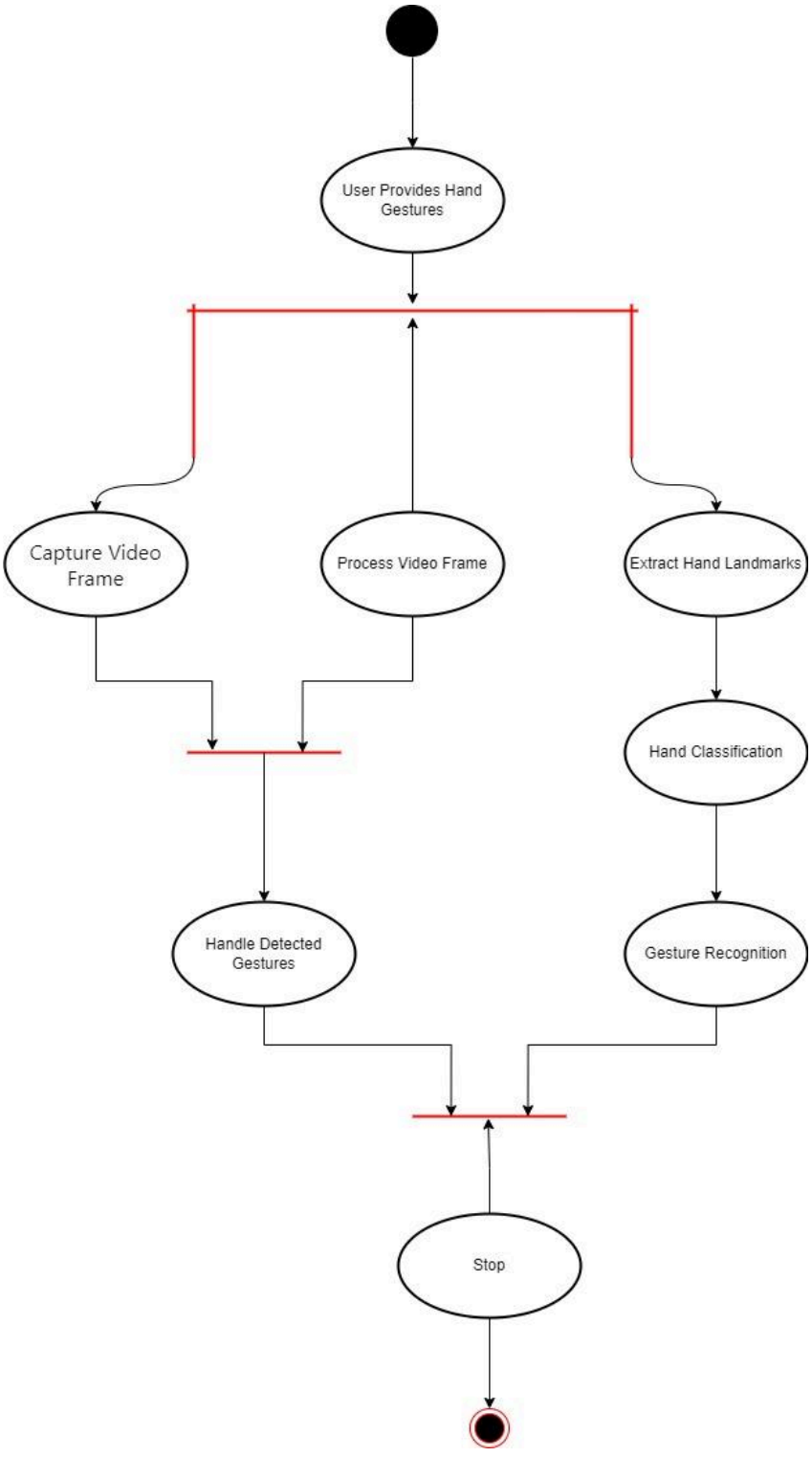
3.3 SEQUENCE DIAGRAM



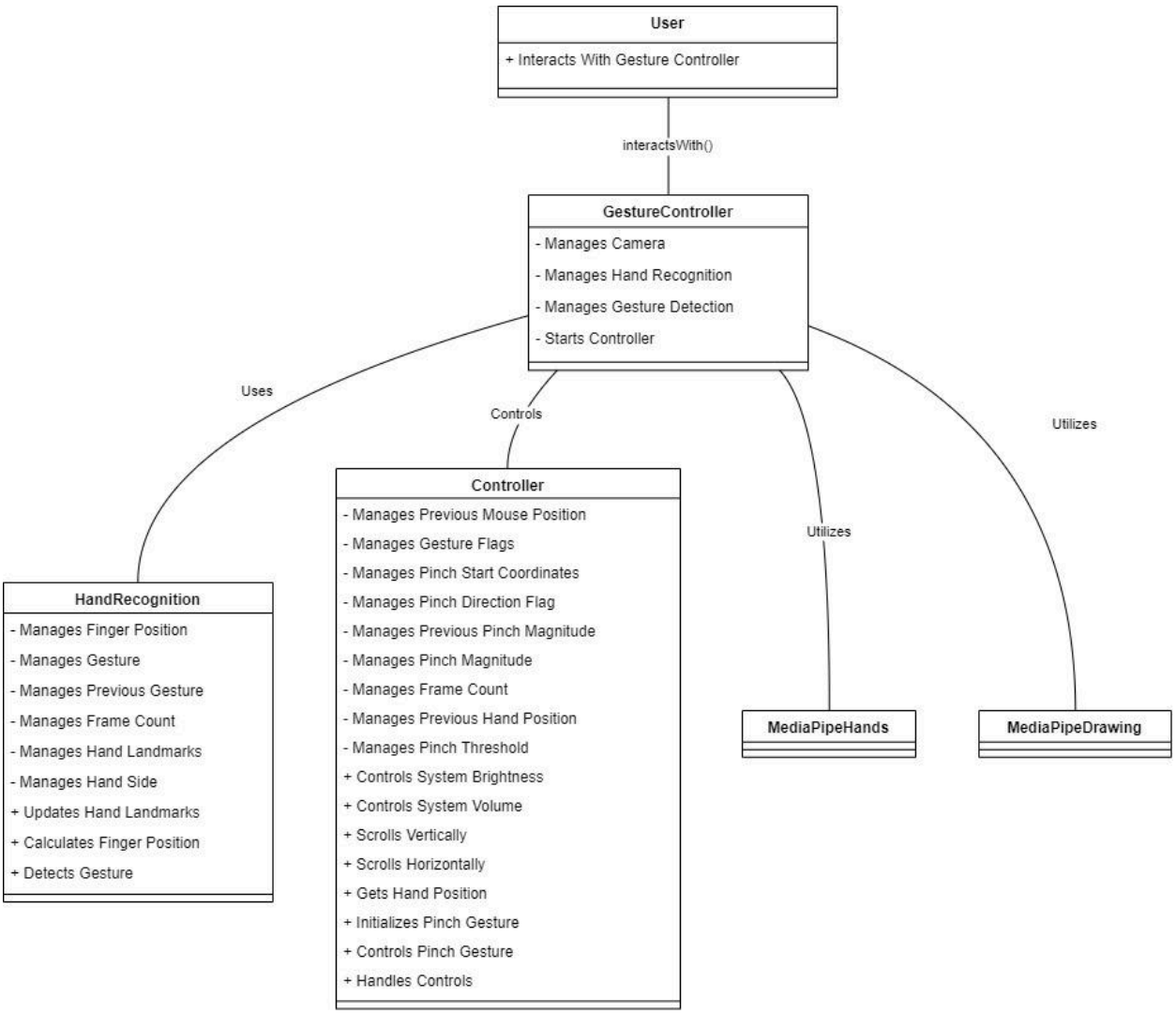
3.4 COLLABORATION DIAGRAM



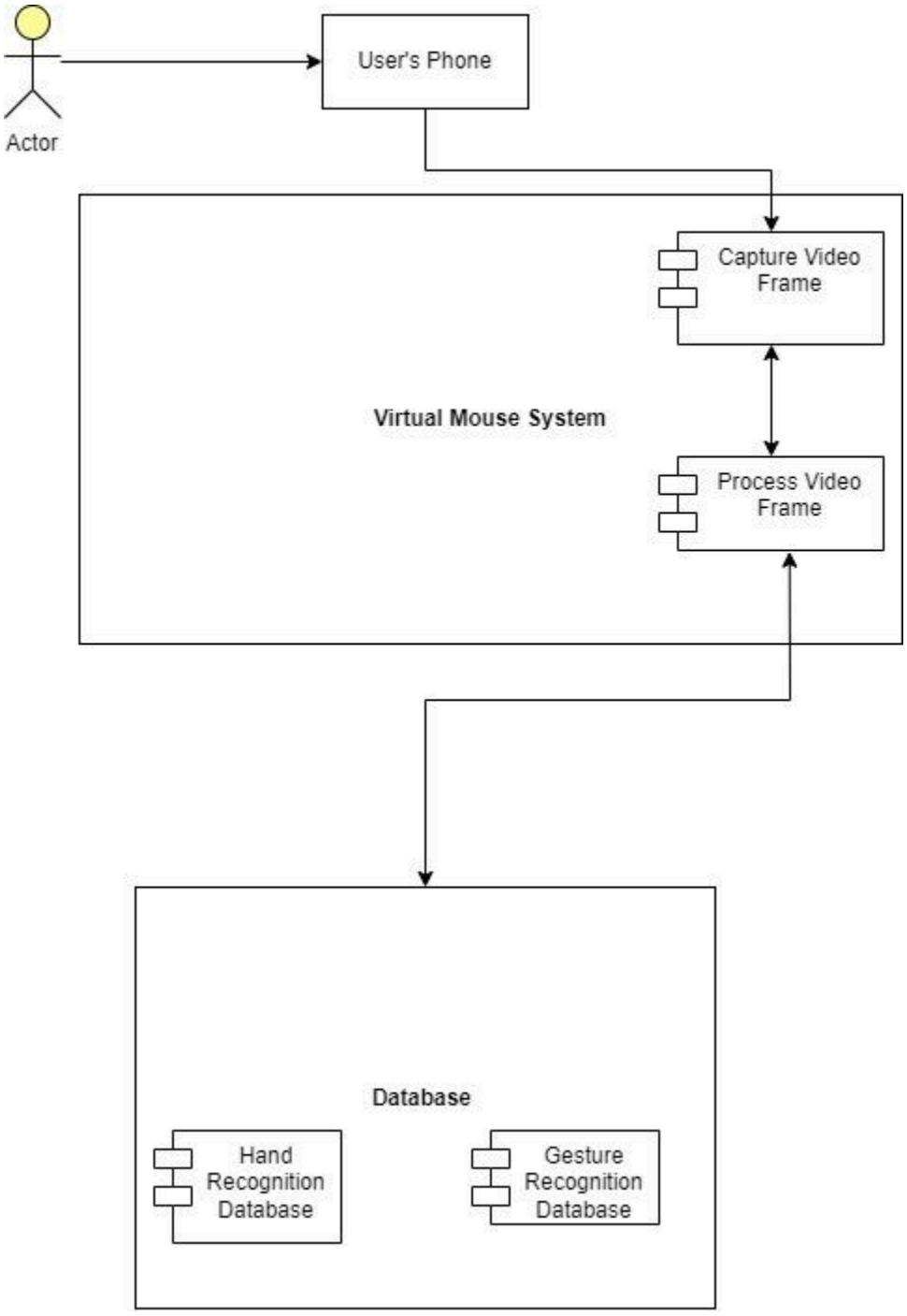
3.5 ACTIVITY DIAGRAM



3.6 COMPONENT DIAGRAM



3.7 DEPLOYMENT DIAGRAM



CHAPTER 4: SOFTWARE METRICS

4.1 METHOD TO FIND COST OF SOFTWARE

Software Metrics for Cost and Time Estimation :

Software metrics are quantitative measures that enable the evaluation and improvement of the software development process. By analyzing various aspects of the project, such as size, complexity, and quality, software metrics help in estimating the effort, cost, and time required for development. Here are some software metrics commonly used for cost and time estimation:

1. Size-Oriented Metrics

- **Lines of Code (LOC):** The total number of lines of code in the project, excluding comments and blank lines.
- **Function Points (FP):** An indicator of the overall functionality of the software based on user inputs, outputs, inquiries, files, and interfaces.

2. Effort-Oriented Metrics

- **Person-Months (PM):** The amount of effort required to complete the project, typically measured in person-months.
- **Work Breakdown Structure (WBS):** A hierarchical decomposition of the project into tasks, which helps in estimating effort distribution.

3. Quality-Oriented Metrics

- **Defect Density**: The number of defects found per unit of software size, indicating the quality of the code.

- **Code Coverage:** The percentage of code covered by automated tests, which reflects the thoroughness of testing efforts.

4. Complexity-Oriented Metrics

- **Cyclomatic Complexity:** A measure of the complexity of the software's control flow, indicating the number of independent paths through the code.
- **Halstead Complexity Measures**: Measures of program complexity based on the number of operators and operands used in the code.

4.2 METHOD APPLIED TO FIND THE COST OF SOFTWARE

Method Name: Function Point Analysis

- External Inputs (EI) :

User Input Risk Assessment = 3

User Input Growth Strategy = 3

- External Outputs (EO) :

Generated Risk Assessment Reports = 4

Growth Strategy Recommendations = 4

- External Inquiries (EQ) :

User Queries Risk Assessment = 3

User Queries Growth Strategy = 3

Now let's assign complexity weights:

- Low Complexity Total Weight (L) : 30
- Average Complexity Total Weight (A) : 16
- High Complexity Total Weight (H) : 48

- Calculating Function Points :

The formula provided is:

FP = (EI x L) + (EO x L) + (EQ x L) +(ILF x A) + (EIF x A)

1. External Inputs (EI):

EI = 3 + 3 = 6

EI x L = 6 x 30 = 180

2. External Outputs (EO):

EO = 4 + 4 = 8

EO x L = 8 x 30 = 240

3. External Inquiries (EQ):

EQ = 3 + 3 = 6

EQ x L = 6 x 30 = 180

4. Internal Logical Files (ILF):

We assume ILF count is 0 for this calculation as it's not given.

5. External Interface Files (EIF):

We assume EIF count is 0 for this calculation as it's not given.

FP = (6 x 30) + (8 x 30) + (6 x 30) + (0 x 16) + (0 x 16)

FP = 180 + 240 + 180 + 0 + 0

FP = 600

Now, based on historical data or industry benchmarks, we can convert these Function Points into effort hours or cost estimates. Let's assume a productivity factor of 25 hours per Function Point:

Effort Hours = FP * 5

Effort Hours = 600* 5

Effort Hours = 3,000

Assuming an hourly rate of $10 for the development team, we can calculate the cost estimate:

Cost = Effort Hours * Hourly Rate

Cost = 3,000* $ 10

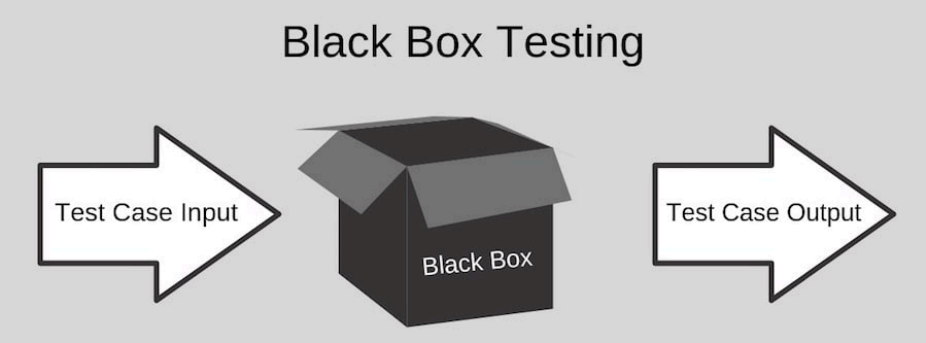
Cost = $ 30,000

So, based on the estimated effort hours and the assumed hourly rate, the cost estimate for developing the "Real-Time Hand Gesture and Control System using Python" would be approximately $ 30,000 .

CHAPTER 5: SOFTWARE TESTING

5.1 Black Box testing of any GUI

Black box testing is a testing technique in which the tester does not know how the software works internally. The tester focuses solely on the software's input and output.



- Test cases for black box testing :

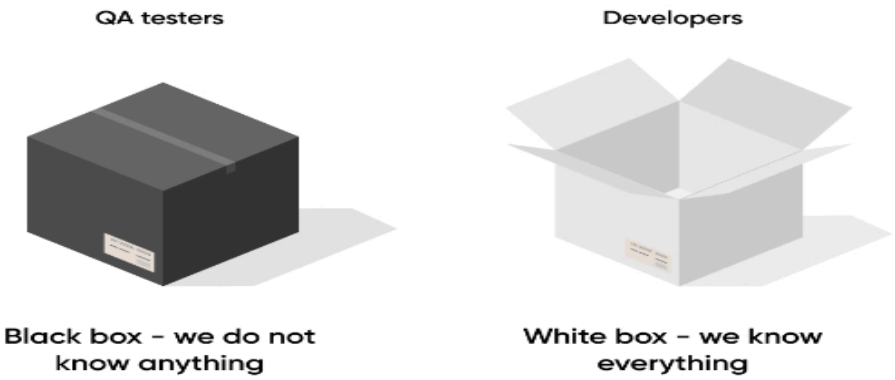
Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC001	Start the application	Run the script	Camera opens, video feed starts, and GUI window displays the video feed	Camera opened, video feed started, and GUI window displayed the video feed	Pass

TC002	Detect major hand gestures	Show right hand with various gestures	Application correctly identifies gestures (FIST, V_GEST, INDEX, etc.) and executes corresponding actions	Right hand gestures detected correctly and correspondin g actions executed	Pass
TC003	Detect minor hand gestures	Show left hand with various gestures	Application correctly identifies gestures (FIST, V_GEST, INDEX, etc.) and executes corresponding actions	Left hand gestures detected correctly and correspondin g actions executed	Pass
TC004	Mouse move with V_GEST	Show V_GEST (major hand)	Cursor moves to the hand position	Cursor moved to the hand position	Pass
TC005	Mouse click with MID	Show MID gesture (major hand)	Mouse click at the hand position	Mouse click at the hand position	Pass
TC006	Right-click with INDEX	Show INDEX gesture (major hand)	Mouse right-click at the hand position	Mouse right-click at the hand position	Pass

TC007	Double-click with TWO_FINGER_CLOSED	Show TWO_FINGER_CLOSED gesture (major hand)	Mouse double-click at the hand position	Mouse double-click at the hand position	Pass
TC008	Drag with FIST	Show FIST gesture (major hand)	Mouse drags from the initial hand position to the new hand position	Mouse dragged from the initial hand position to the new hand position	Pass
TC009	Scroll vertically with PINCH_MINOR	Show PINCH_MINOR gesture (minor hand)	Vertical scroll occurs on the screen	Vertical scroll occurred on the screen	Pass
TC010	Scroll horizontally with PINCH_MINOR	Move PINCH_MINOR gesture horizontally	Horizontal scroll occurs on the screen	Horizontal scroll occurred on the screen	Pass
TC011	Close application	Click “X”	Application closes and releases the camera	Application closed and camera released	Pass

1.1 White box testing of any running code.

Unlike black box testing, which focuses solely on software functionality, white box testing methods examine the internal architecture, code structure, employed data structures, and overall software functioning. It is also known as structural testing, glass box testing, or clear box testing. Transparent testing or open box testing are other names for white box testing.



Code snippet 1:

```
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
```

Code snippet 2:

```
    def get_dz(self, point):

        return abs(self.hand_result.landmark[point[0]].z -
self.hand_result.landmark[point[1]].z)
    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


    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
```

Code snippet 3:

```
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():

        currentBrightnessLv = sbcontrol.get_brightness(display=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=sbcontrol.get_brightness(display=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():

        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')

    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):
```

❖ White box testing on the code snippet for "Real-Time Hand Gesture Recognition and Control System using Python" :

1. Input Validation Testing

- Validate valid input gestures like FIST, V_GEST, INDEX.

- Test multiple gestures in sequence.

- Test invalid input values like undefined gestures or random movements, objects, or non-hand entities.

- Test mixed gestures and transitions.

2. Functionality Testing

- Test individual functions for gesture detection, control functions (mouse move, click, scroll), and application functions (start and close).

3. Data Processing Testing

- Test data processing with various input data like different lighting conditions, backgrounds, and multiple hands with different skin tones.

4. Error Handling Testing

- Test error scenarios like camera access issues, unexpected gestures or positions, and high system load.

5. Integration Testing

- Test camera integration, gesture detection, and system control integration points.

6. Performance Testing

- Test scalability and efficiency with continuous input over time and measure response time for real-time performance.

7. Security Testing

- Test camera access security, data exposure, and proper handling of control commands.

8. Boundary Testing

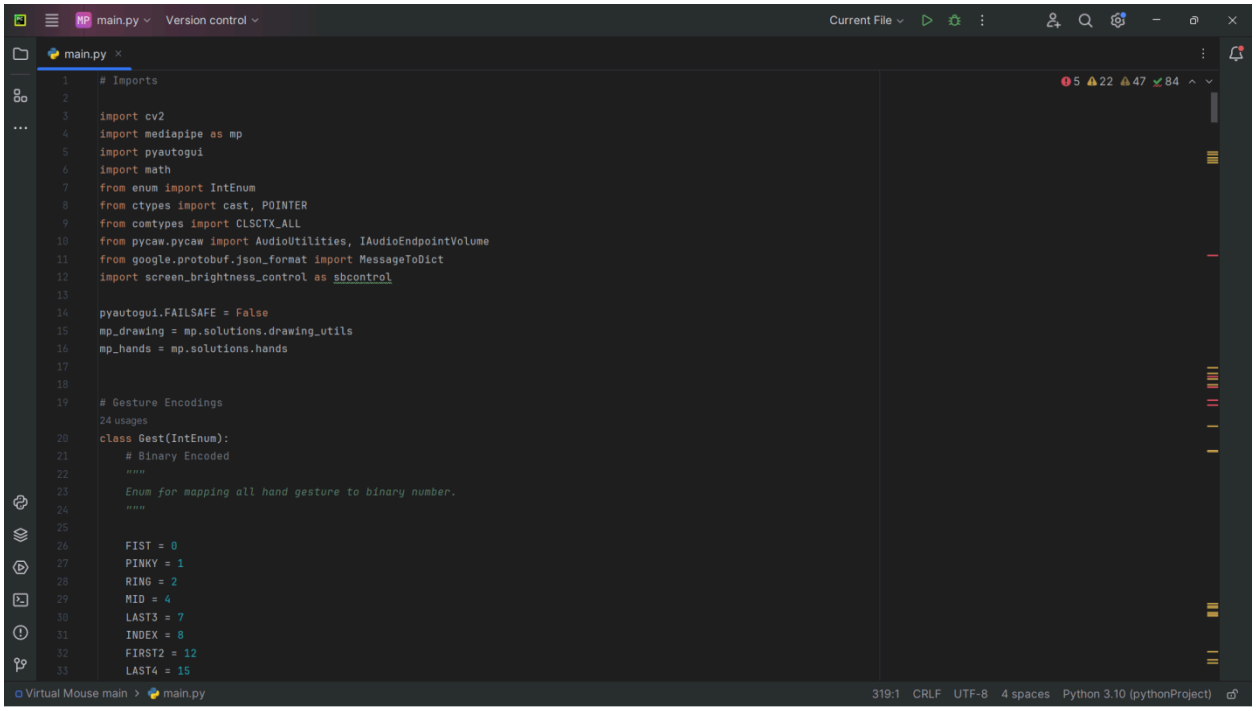
- Test minimal and exaggerated movements and varying hand sizes and distances.

9. Concurrency Testing

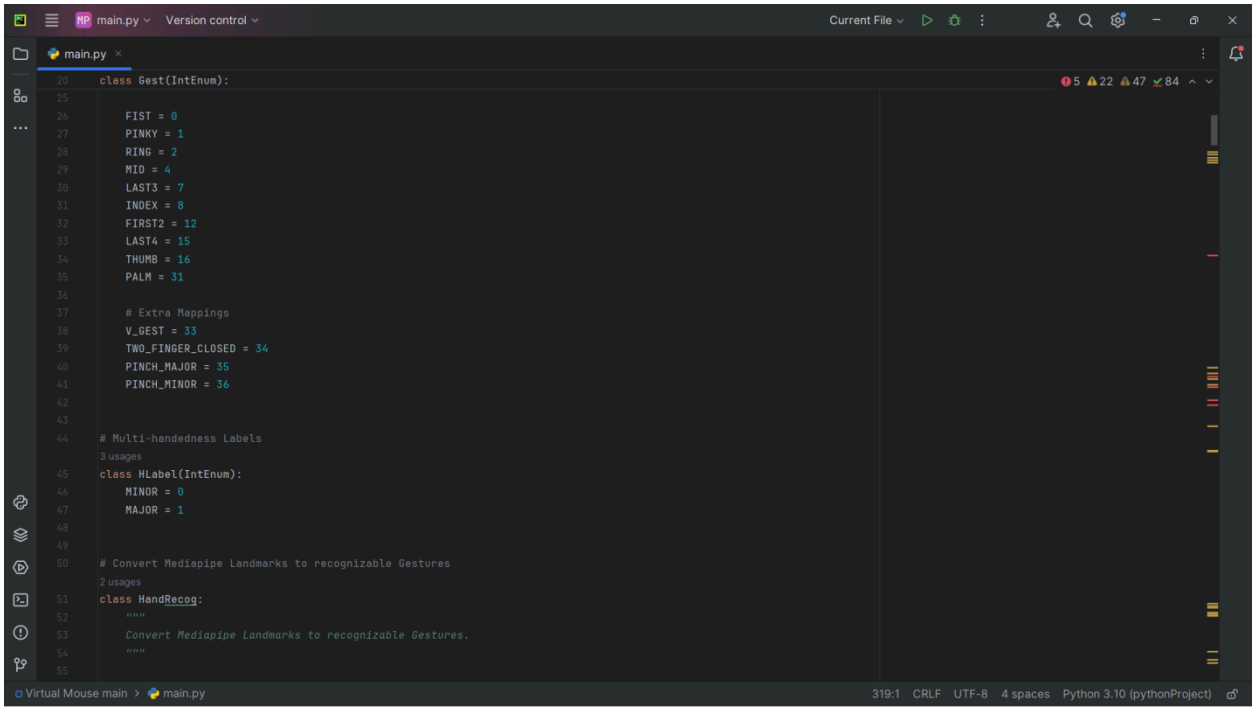
- Test multiple simultaneous gestures and multiple user inputs (if applicable).

CHAPTER 6: PROJECT CODE AND OUTPUT

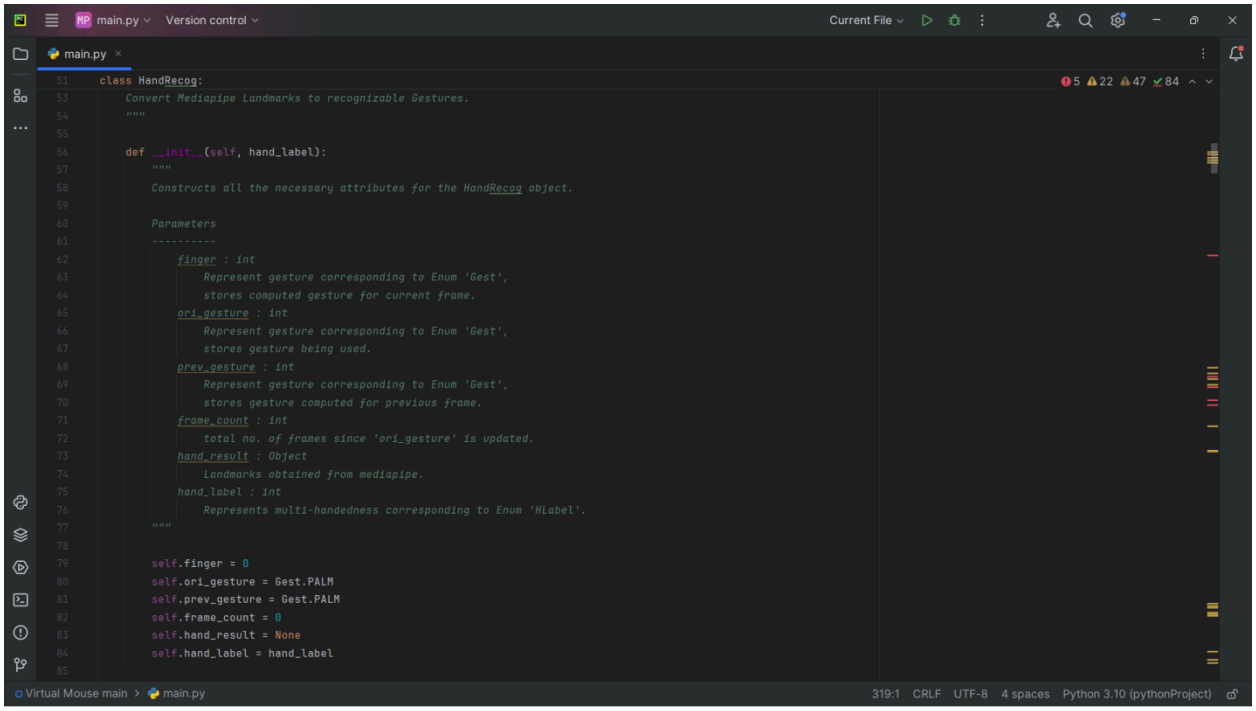
6.1 Running code:



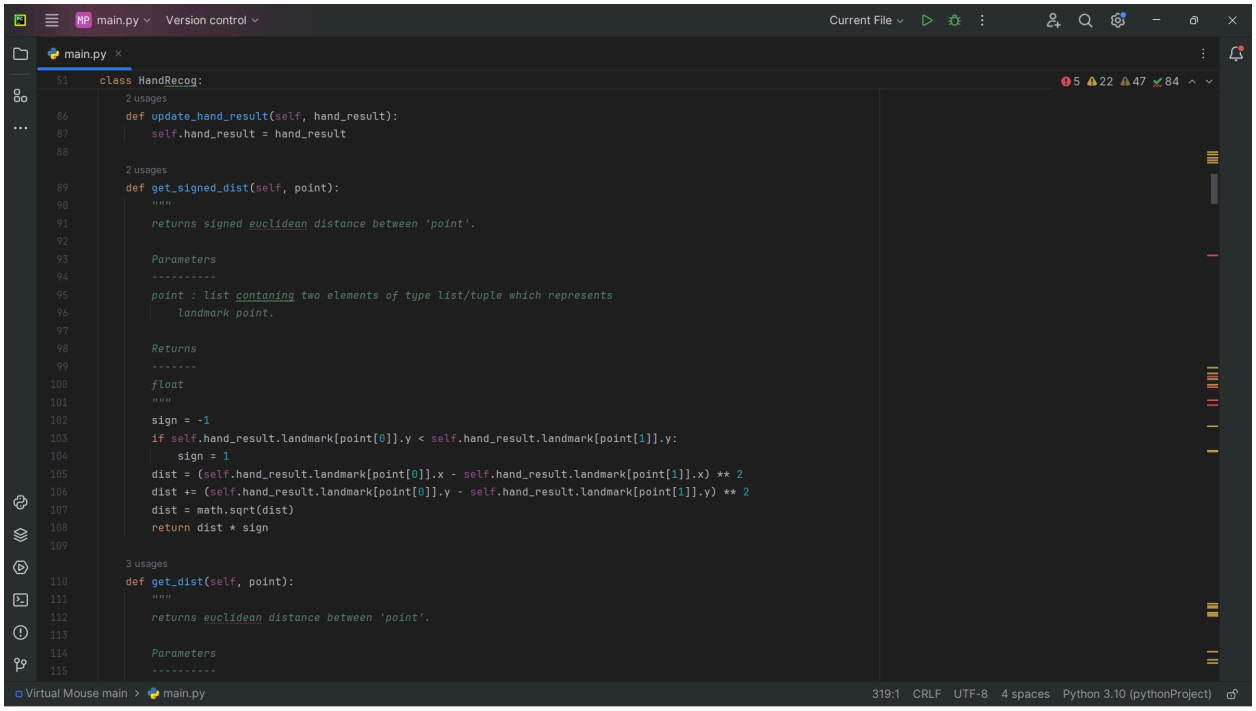
```
# Imports


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

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


# Gesture Encodings
24 usages
class Gest(IntEnum):
    # Binary Encoded
    """
    Enum for mapping all hand gesture to binary number.
    """

    FIST = 0
    PINKY = 1
    RING = 2
    MID = 4
    LAST3 = 7
    INDEX = 8
    FIRST2 = 12
    LAST4 = 15
```



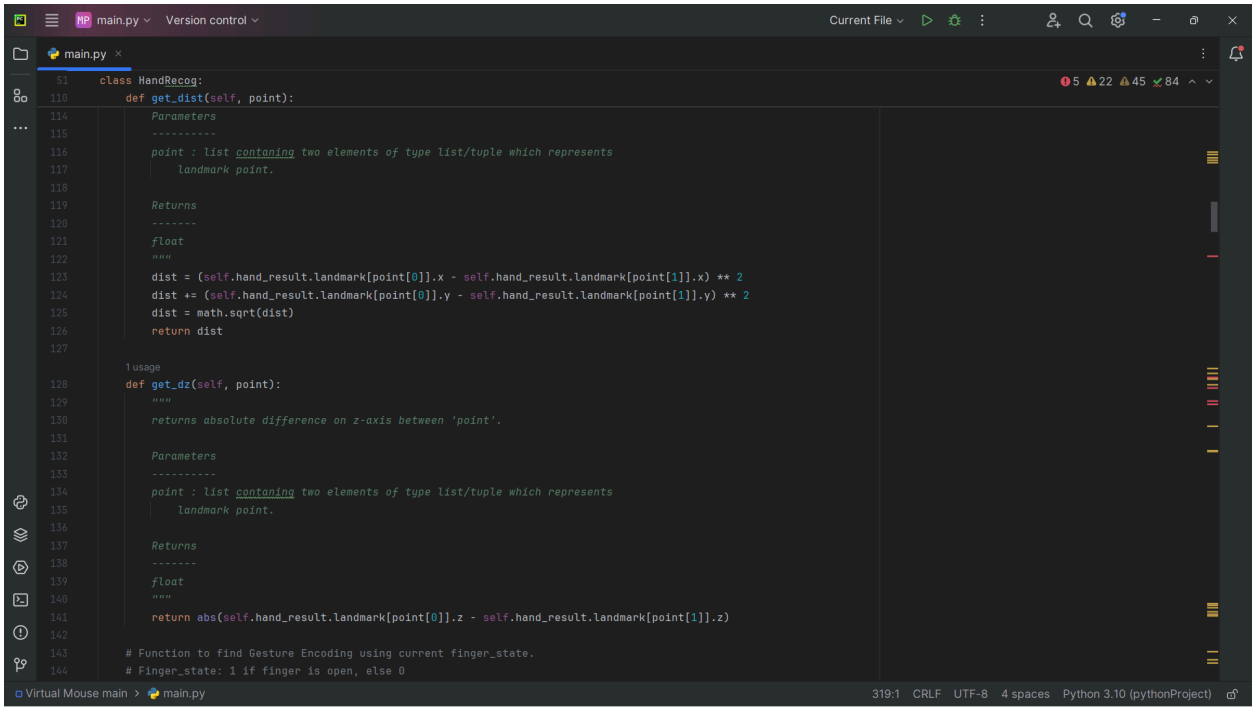
```python
class Gest(IntEnum):

    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
3 usages
class HLabel(IntEnum):
    MINOR = 0
    MAJOR = 1


# Convert Mediapipe Landmarks to recognizable Gestures
2 usages
class HandRecog:
    """
    Convert Mediapipe Landmarks to recognizable Gestures.
    """
```



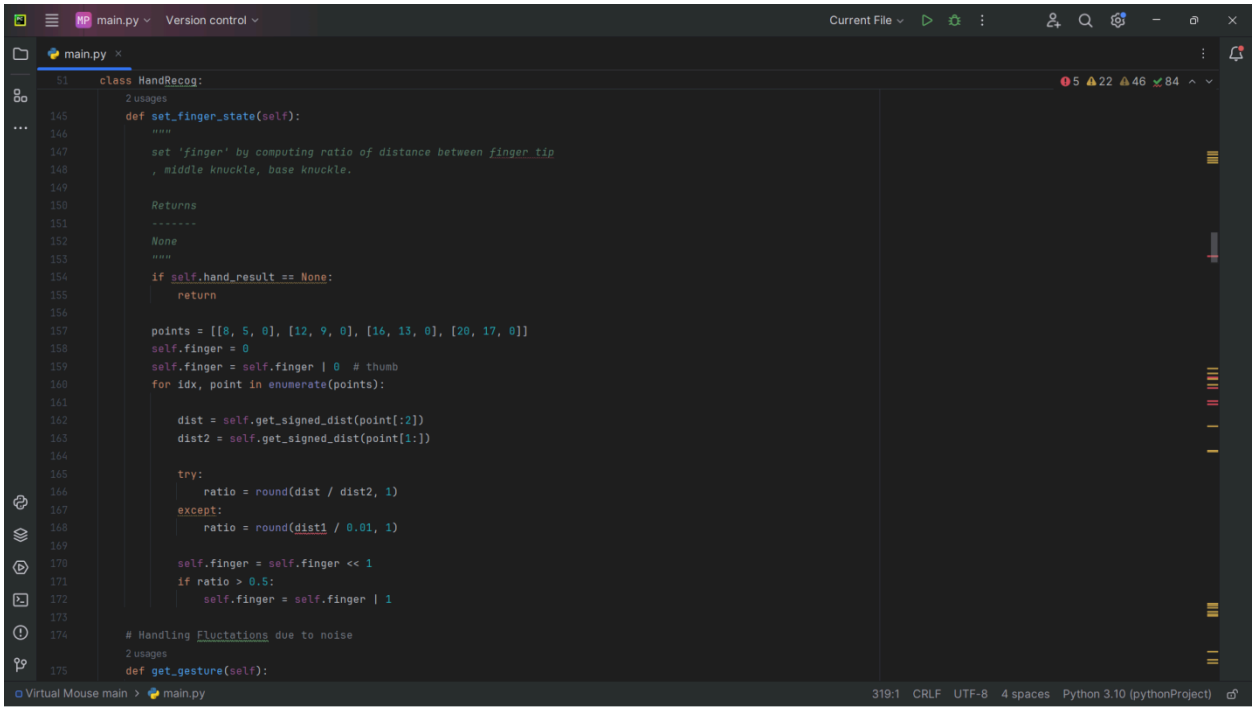
```python
class HandRecog:
    Convert Mediapipe Landmarks to recognizable Gestures.
    """

    def __init__(self, hand_label):
        """
        Constructs all the necessary attributes for the HandRecog object.

        Parameters
        ----------
            finger : int
                Represent gesture corresponding to Enum 'Gest',
                stores computed gesture for current frame.
            ori_gesture : int
                Represent gesture corresponding to Enum 'Gest',
                stores gesture being used.
            prev_gesture : int
                Represent gesture corresponding to Enum 'Gest',
                stores gesture computed for previous frame.
            frame_count : int
                total no. of frames since 'ori_gesture' is updated.
            hand_result : Object
                Landmarks obtained from mediapipe.
            hand_label : int
                Represents multi-handedness corresponding to Enum 'HLabel'.
        """

        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
```



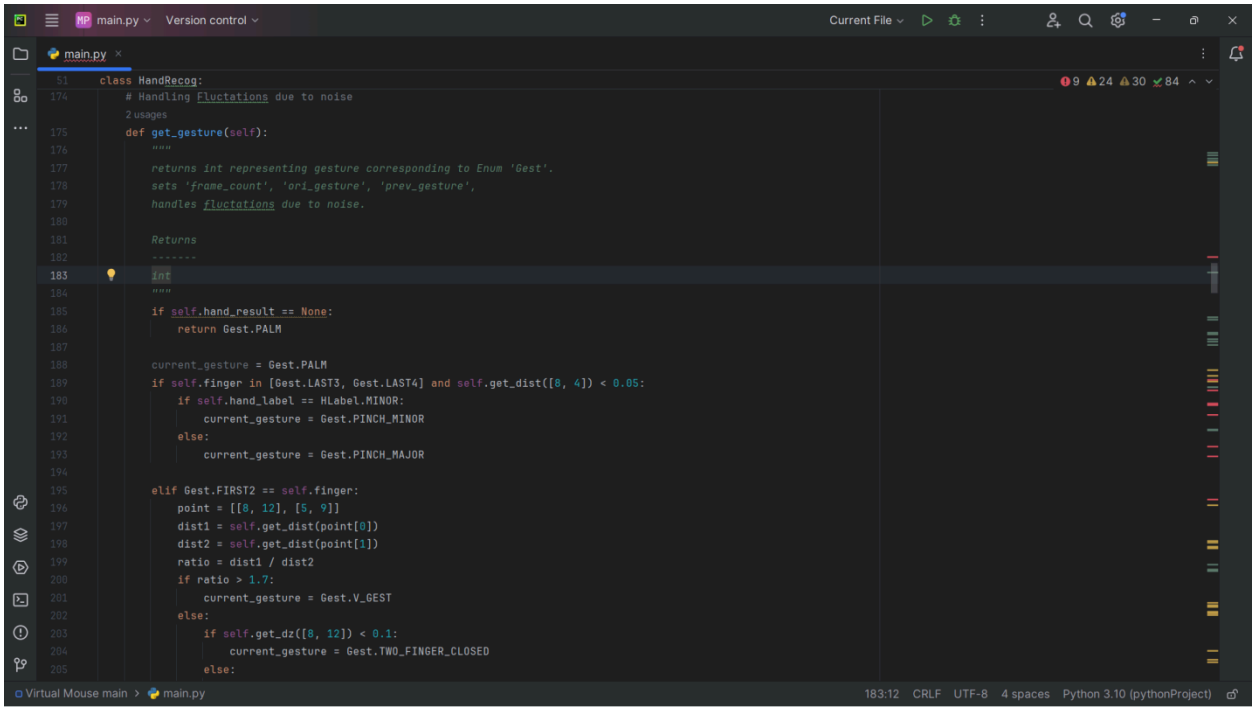
```python
class HandRecog:
    2 usages
    def update_hand_result(self, hand_result):
        self.hand_result = hand_result

    2 usages
    def get_signed_dist(self, point):
        """
        returns signed euclidean distance between 'point'.

        Parameters
        ----------
        point : list contaning two elements of type list/tuple which represents
            landmark point.

        Returns
        -------
        float
        """
        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

    3 usages
    def get_dist(self, point):
        """
        returns euclidean distance between 'point'.

        Parameters
        ----------
```



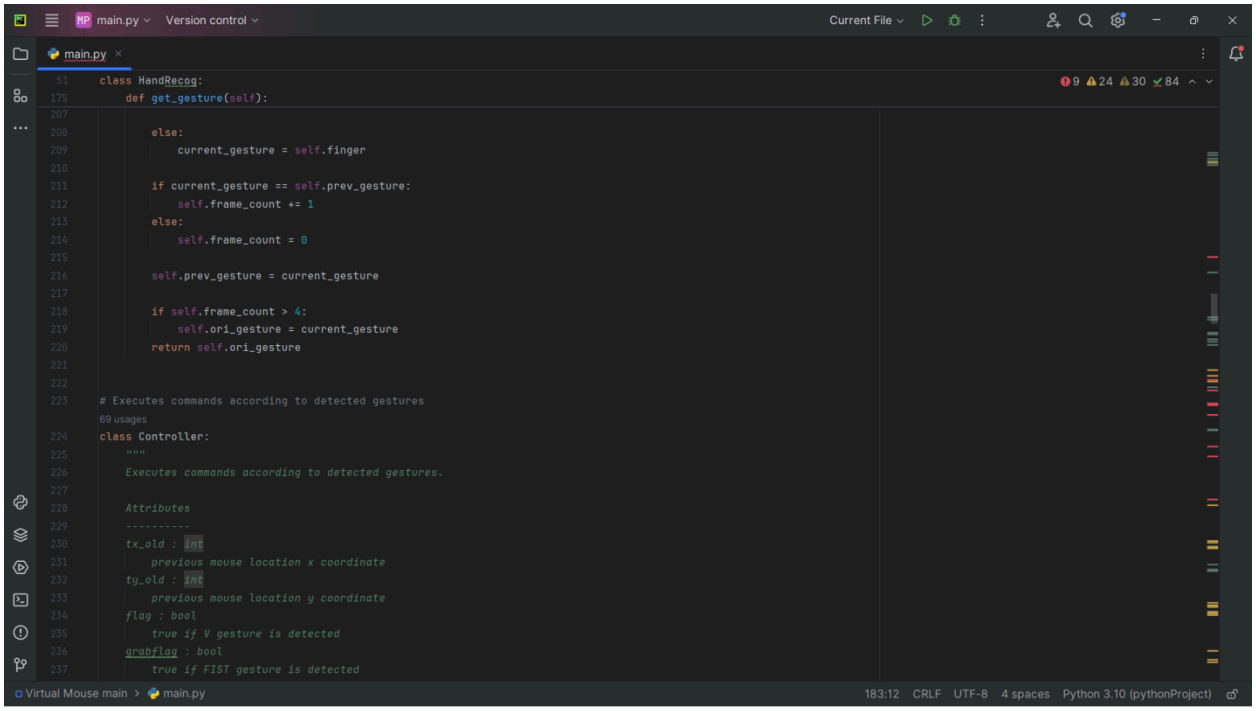
```python
class HandRecog:
    def get_dist(self, point):
        Parameters
        ----------
        point : list contaning two elements of type list/tuple which represents
            landmark point.

        Returns
        -------
        float
        """
        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

    1 usage
    def get_dz(self, point):
        """
        returns absolute difference on z-axis between 'point'.

        Parameters
        ----------
        point : list contaning two elements of type list/tuple which represents
            landmark point.

        Returns
        -------
        float
        """
        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
```



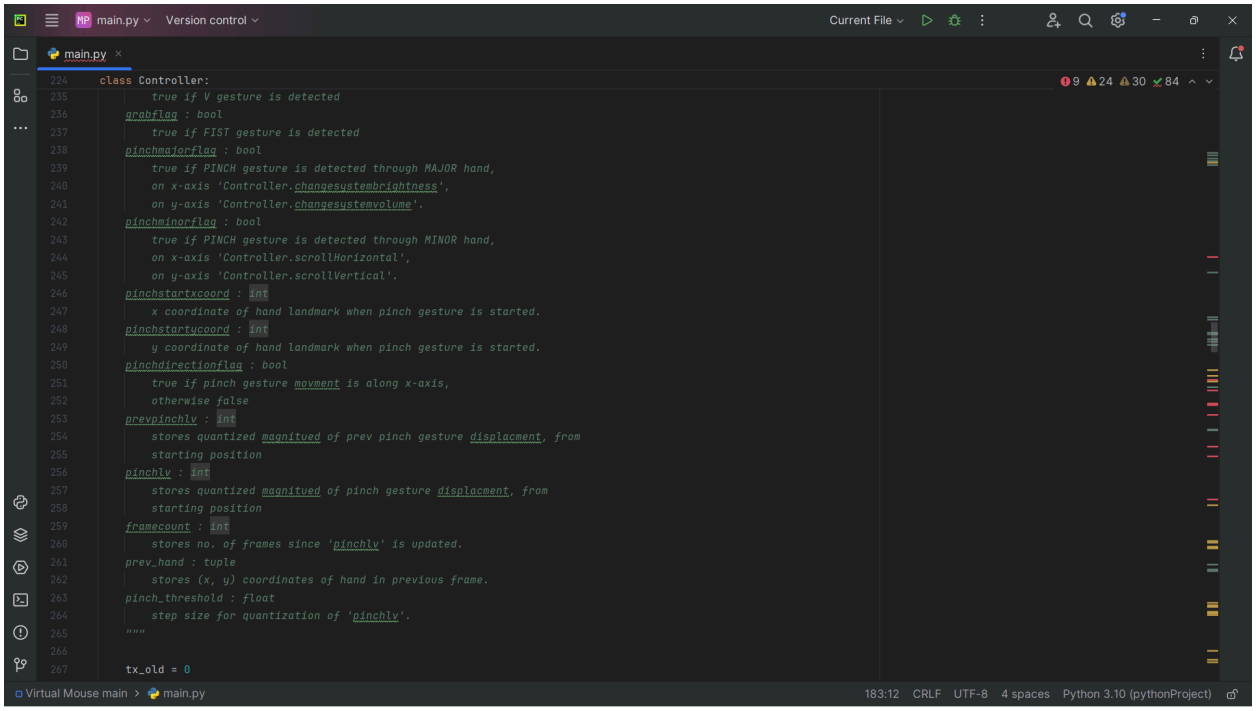
```python
class HandRecog:
    2 usages
    def set_finger_state(self):
        """
        set 'finger' by computing ratio of distance between finger tip
        , middle knuckle, base knuckle.

        Returns
        -------
        None
        """
        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
    2 usages
    def get_gesture(self):
```



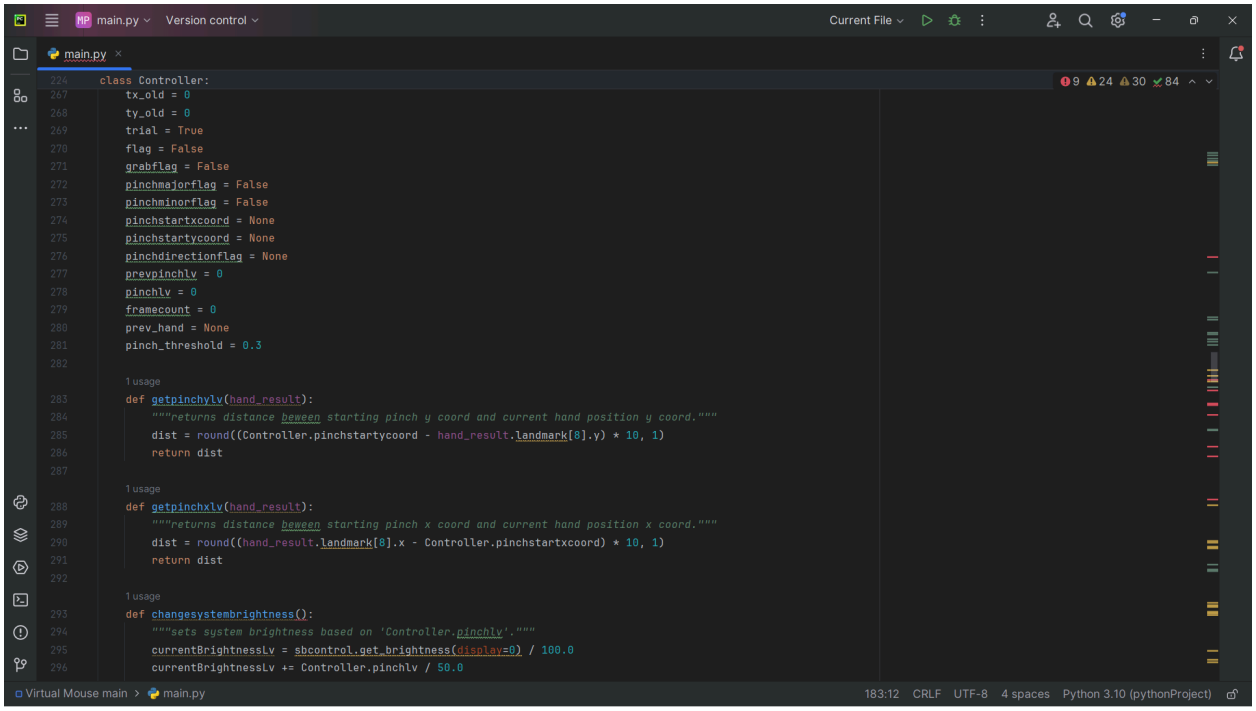
```python
class HandRecog:
    # Handling Fluctations due to noise
    2 usages
    def get_gesture(self):
        """
        returns int representing gesture corresponding to Enum 'Gest'.
        sets 'frame_count', 'ori_gesture', 'prev_gesture',
        handles fluctations due to noise.

        Returns
        -------
        int
        """
        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:
```



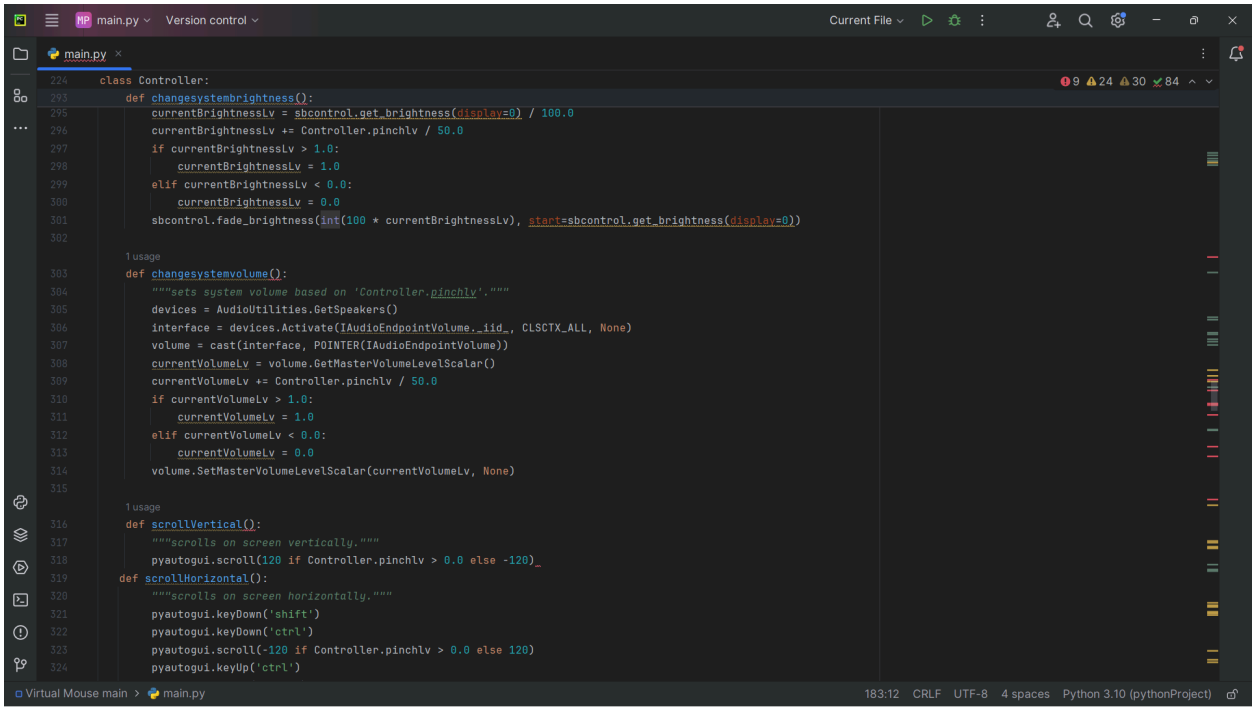
```
class HandRecog:
    def get_gesture(self):

        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


# Executes commands according to detected gestures
class Controller:
    """
    Executes commands according to detected gestures.

    Attributes
    ----------
    tx_old : int
        previous mouse location x coordinate
    ty_old : int
        previous mouse location y coordinate
    flag : bool
        true if V gesture is detected
    grabflag : bool
        true if FIST gesture is detected
```



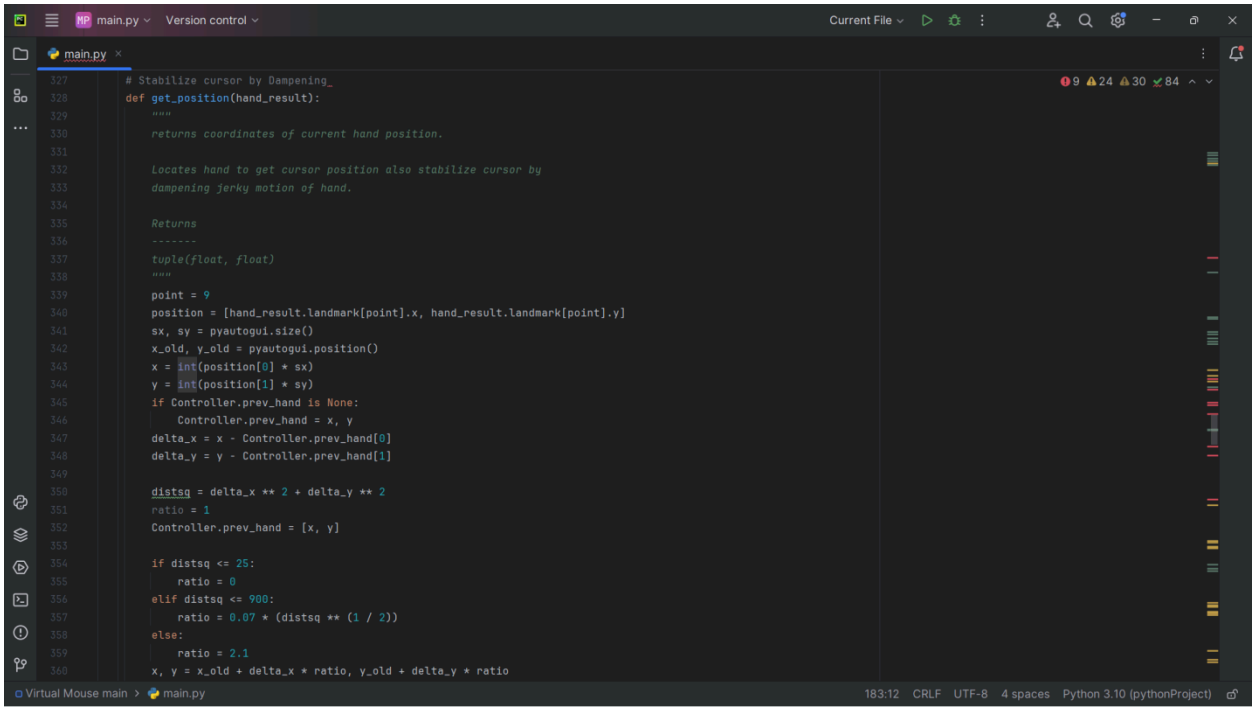
```
class Controller:
        true if V gesture is detected
    grabflag : bool
        true if FIST gesture is detected
    pinchmajorflag : bool
        true if PINCH gesture is detected through MAJOR hand,
        on x-axis 'Controller.changesystembrightness',
        on y-axis 'Controller.changesystemvolume'.
    pinchminorflag : bool
        true if PINCH gesture is detected through MINOR hand,
        on x-axis 'Controller.scrollHorizontal',
        on y-axis 'Controller.scrollVertical'.
    pinchstartxcoord : int
        x coordinate of hand landmark when pinch gesture is started.
    pinchstartycoord : int
        y coordinate of hand landmark when pinch gesture is started.
    pinchdirectionflag : bool
        true if pinch gesture movment is along x-axis,
        otherwise false
    prevpinchlv : int
        stores quantized magnitued of prev pinch gesture displacment, from
        starting position
    pinchlv : int
        stores quantized magnitued of pinch gesture displacment, from
        starting position
    framecount : int
        stores no. of frames since 'pinchlv' is updated.
    prev_hand : tuple
        stores (x, y) coordinates of hand in previous frame.
    pinch_threshold : float
        step size for quantization of 'pinchlv'.
    """

    tx_old = 0
```



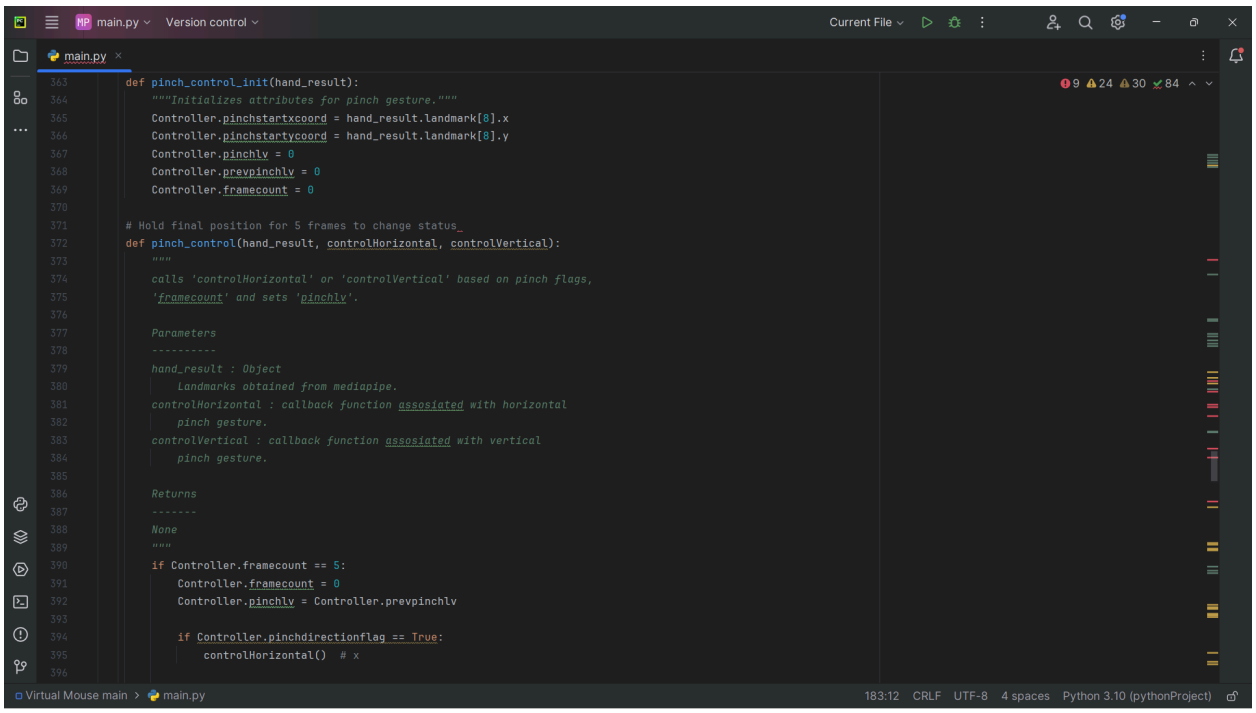
```python
class Controller:
    tx_old = 0
    ty_old = 0
    trial = True
    flag = False
    grabflag = False
    pinchmajorflag = False
    pinchminorflag = False
    pinchstartxcoord = None
    pinchstartycoord = None
    pinchdirectionflag = None
    prevpinchlv = 0
    pinchlv = 0
    framecount = 0
    prev_hand = None
    pinch_threshold = 0.3

    1 usage
    def getpinchylv(hand_result):
        """returns distance beween starting pinch y coord and current hand position y coord."""
        dist = round((Controller.pinchstartycoord - hand_result.landmark[8].y) * 10, 1)
        return dist

    1 usage
    def getpinchxlv(hand_result):
        """returns distance beween starting pinch x coord and current hand position x coord."""
        dist = round((hand_result.landmark[8].x - Controller.pinchstartxcoord) * 10, 1)
        return dist

    1 usage
    def changesystembrightness():
        """sets system brightness based on 'Controller.pinchlv'."""
        currentBrightnessLv = sbcontrol.get_brightness(display=0) / 100.0
        currentBrightnessLv += Controller.pinchlv / 50.0
```



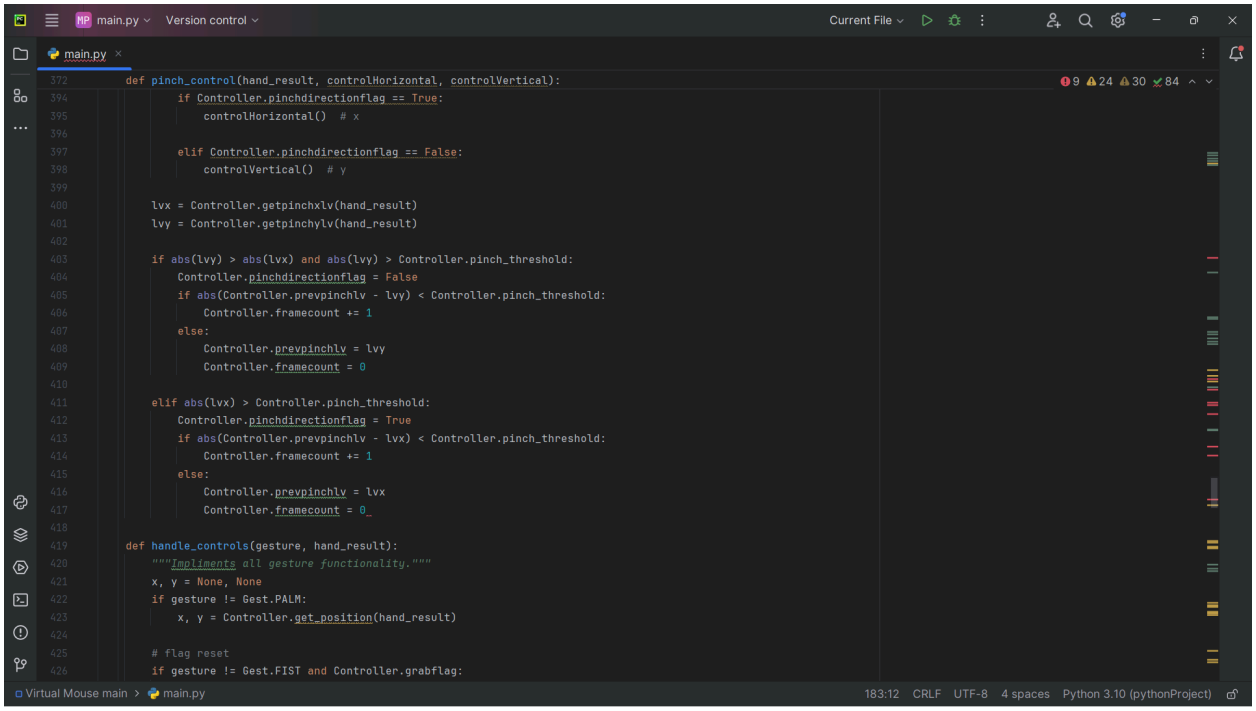
```python
class Controller:
    def changesystembrightness():
        currentBrightnessLv = sbcontrol.get_brightness(display=0) / 100.0
        currentBrightnessLv += Controller.pinchlv / 50.0
        if currentBrightnessLv > 1.0:
            currentBrightnessLv = 1.0
        elif currentBrightnessLv < 0.0:
            currentBrightnessLv = 0.0
        sbcontrol.fade_brightness(int(100 * currentBrightnessLv), start=sbcontrol.get_brightness(display=0))

    1 usage
    def changesystemvolume():
        """sets system volume based on 'Controller.pinchlv'."""
        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)

    1 usage
    def scrollVertical():
        """scrolls on screen vertically."""
        pyautogui.scroll(120 if Controller.pinchlv > 0.0 else -120)
    def scrollHorizontal():
        """scrolls on screen horizontally."""
        pyautogui.keyDown('shift')
        pyautogui.keyDown('ctrl')
        pyautogui.scroll(-120 if Controller.pinchlv > 0.0 else 120)
        pyautogui.keyUp('ctrl')
```



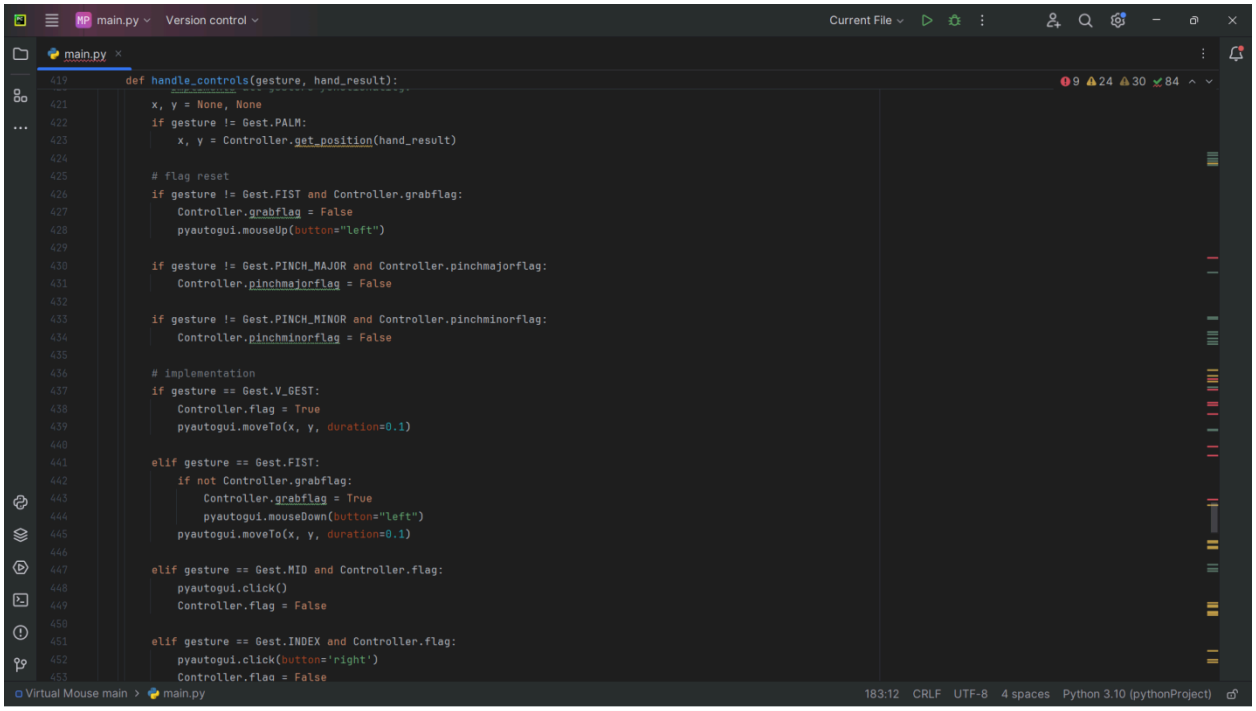
```python
# Stabilize cursor by Dampening_
def get_position(hand_result):
    """
    returns coordinates of current hand position.

    Locates hand to get cursor position also stabilize cursor by
    dampening jerky motion of hand.

    Returns
    -------
    tuple(float, float)
    """
    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
```



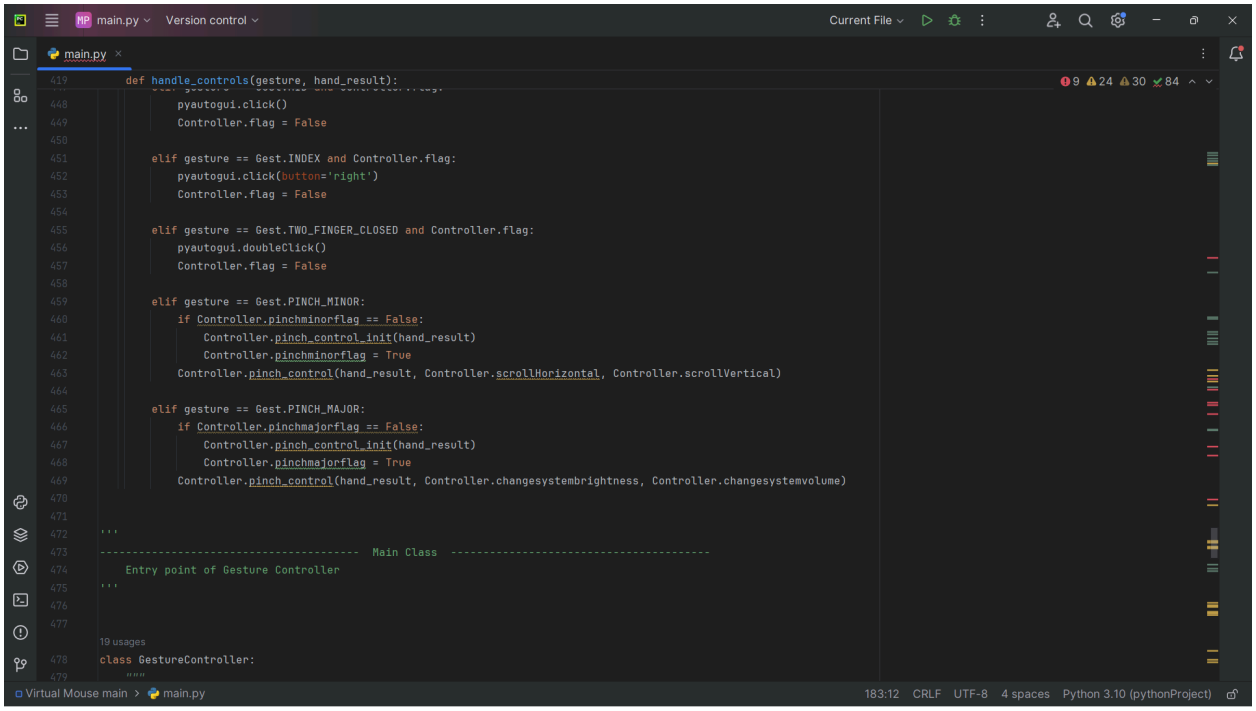
```python
def pinch_control_init(hand_result):
    """Initializes attributes for pinch gesture."""
    Controller.pinchstartxcoord = hand_result.landmark[8].x
    Controller.pinchstartycoord = hand_result.landmark[8].y
    Controller.pinchlv = 0
    Controller.prevpinchlv = 0
    Controller.framecount = 0

# Hold final position for 5 frames to change status_
def pinch_control(hand_result, controlHorizontal, controlVertical):
    """
    calls 'controlHorizontal' or 'controlVertical' based on pinch flags,
    'framecount' and sets 'pinchlv'.

    Parameters
    ----------
    hand_result : Object
        Landmarks obtained from mediapipe.
    controlHorizontal : callback function assosiated with horizontal
        pinch gesture.
    controlVertical : callback function assosiated with vertical
        pinch gesture.

    Returns
    -------
    None
    """
    if Controller.framecount == 5:
        Controller.framecount = 0
        Controller.pinchlv = Controller.prevpinchlv

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



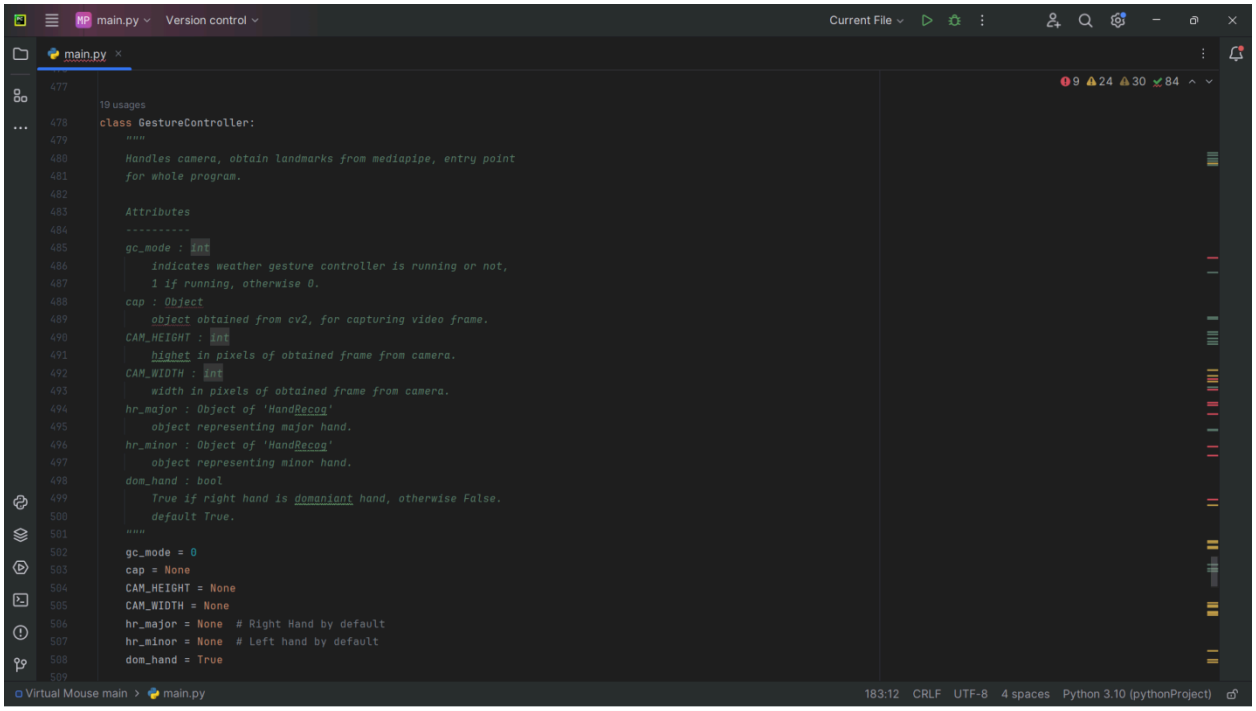
```
def pinch_control(hand_result, controlHorizontal, controlVertical):
            if Controller.pinchdirectionflag == True:
                controlHorizontal()  # x

            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):
        """Impliments all gesture functionality."""
        x, y = None, None
        if gesture != Gest.PALM:
            x, y = Controller.get_position(hand_result)

        # flag reset
        if gesture != Gest.FIST and Controller.grabflag:
```



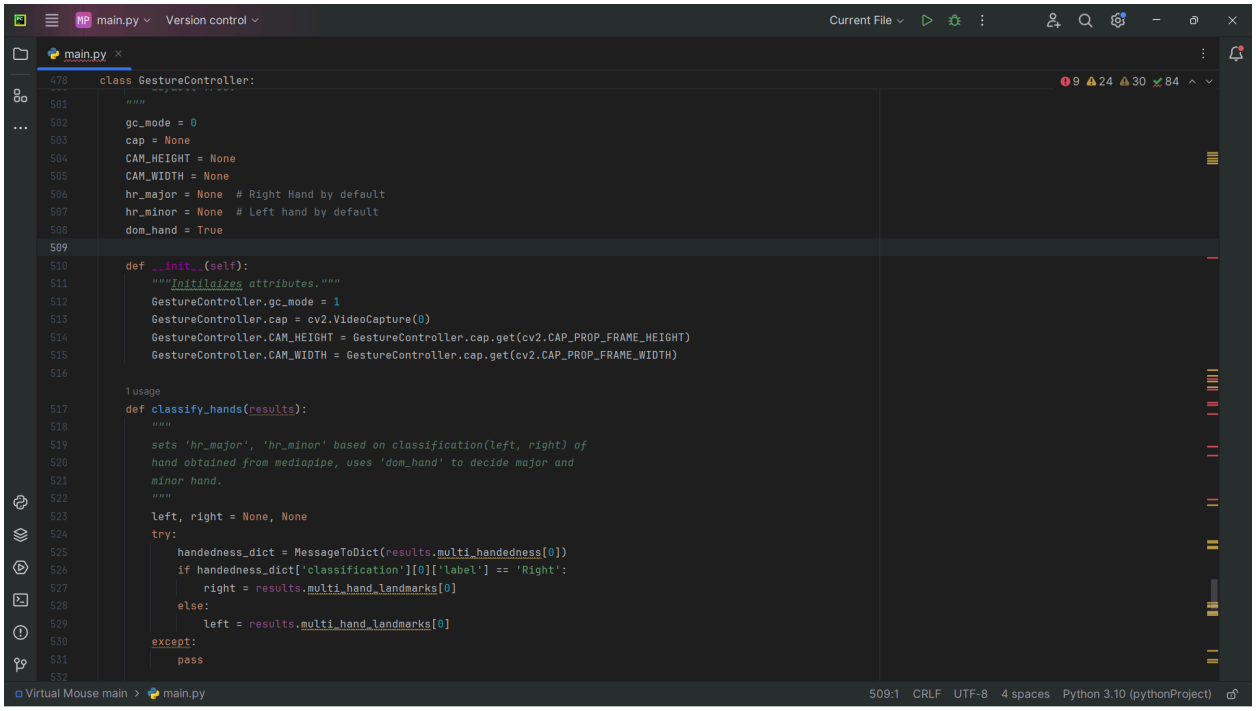
```
def handle_controls(gesture, hand_result):
        x, y = None, None
        if gesture != Gest.PALM:
            x, y = Controller.get_position(hand_result)

        # 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
```



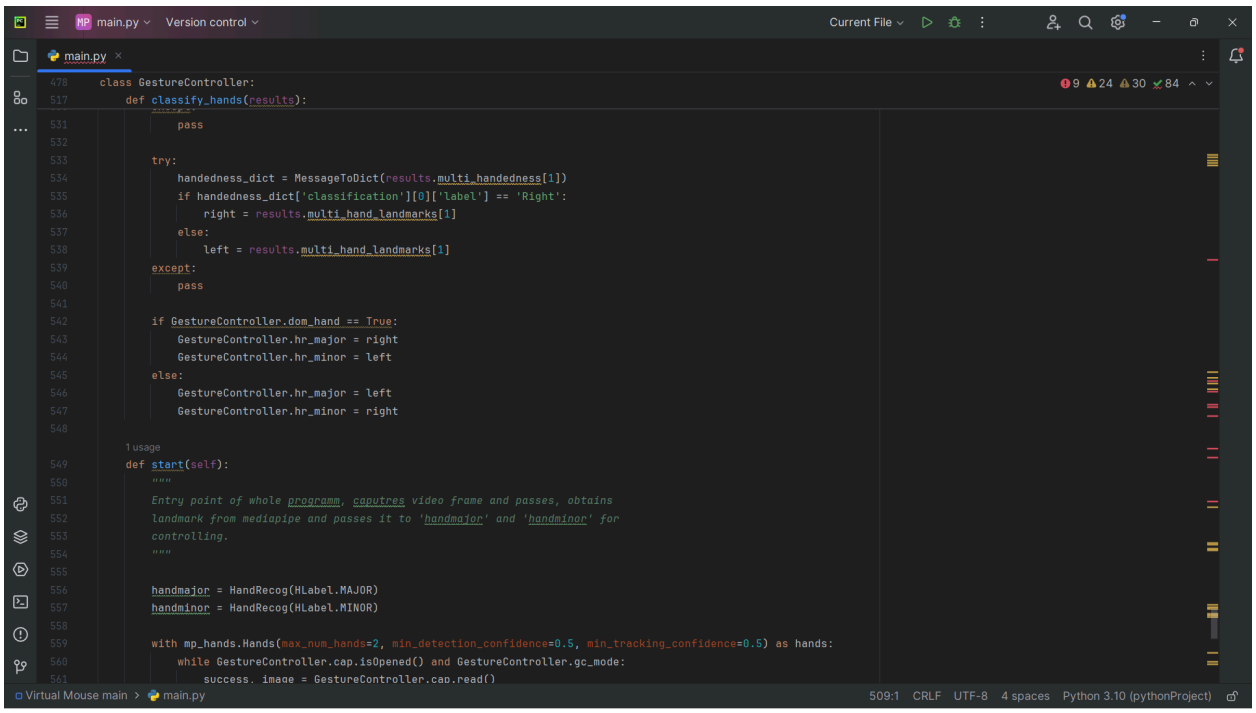
```
        def handle_controls(gesture, hand_result):
                pyautogui.click()
                Controller.flag = False

            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)


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


    19 usages
    class GestureController:
        """
```



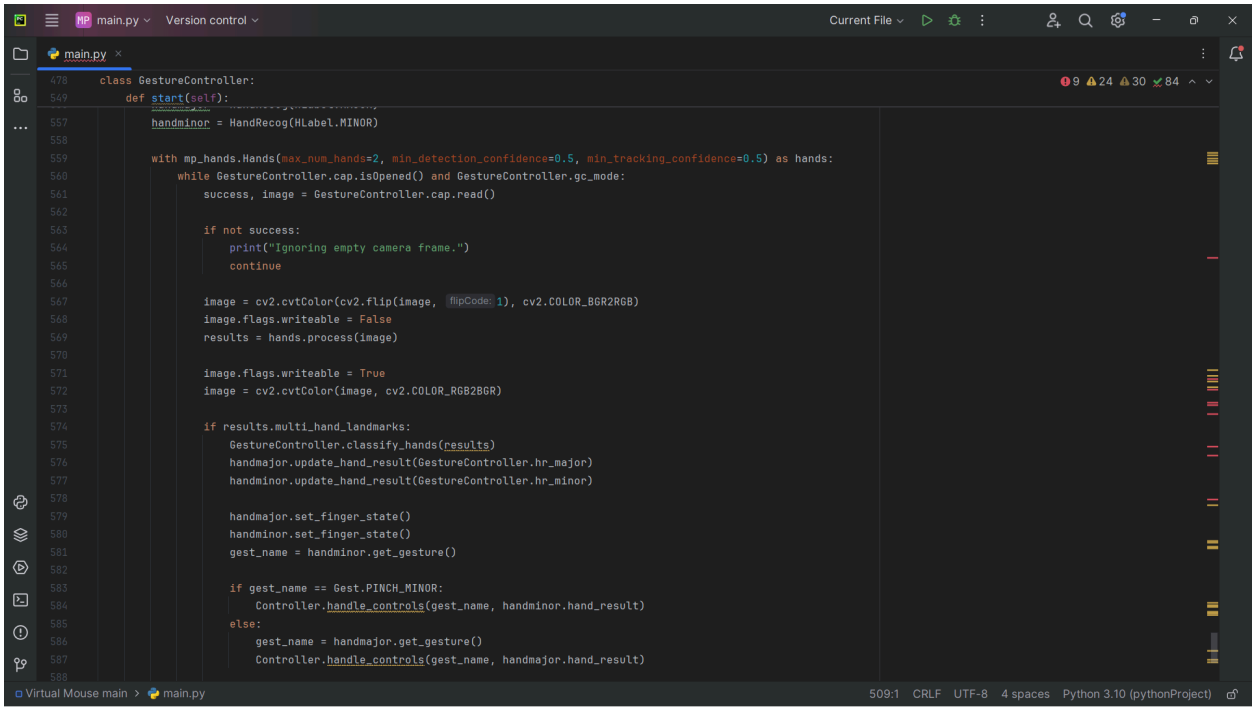
```
    19 usages
    class GestureController:
        """
        Handles camera, obtain landmarks from mediapipe, entry point
        for whole program.

        Attributes
        ----------
        gc_mode : int
            indicates weather gesture controller is running or not,
            1 if running, otherwise 0.
        cap : Object
            object obtained from cv2, for capturing video frame.
        CAM_HEIGHT : int
            highet in pixels of obtained frame from camera.
        CAM_WIDTH : int
            width in pixels of obtained frame from camera.
        hr_major : Object of 'HandRecog'
            object representing major hand.
        hr_minor : Object of 'HandRecog'
            object representing minor hand.
        dom_hand : bool
            True if right hand is domaniant hand, otherwise False.
            default True.
        """
        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
```



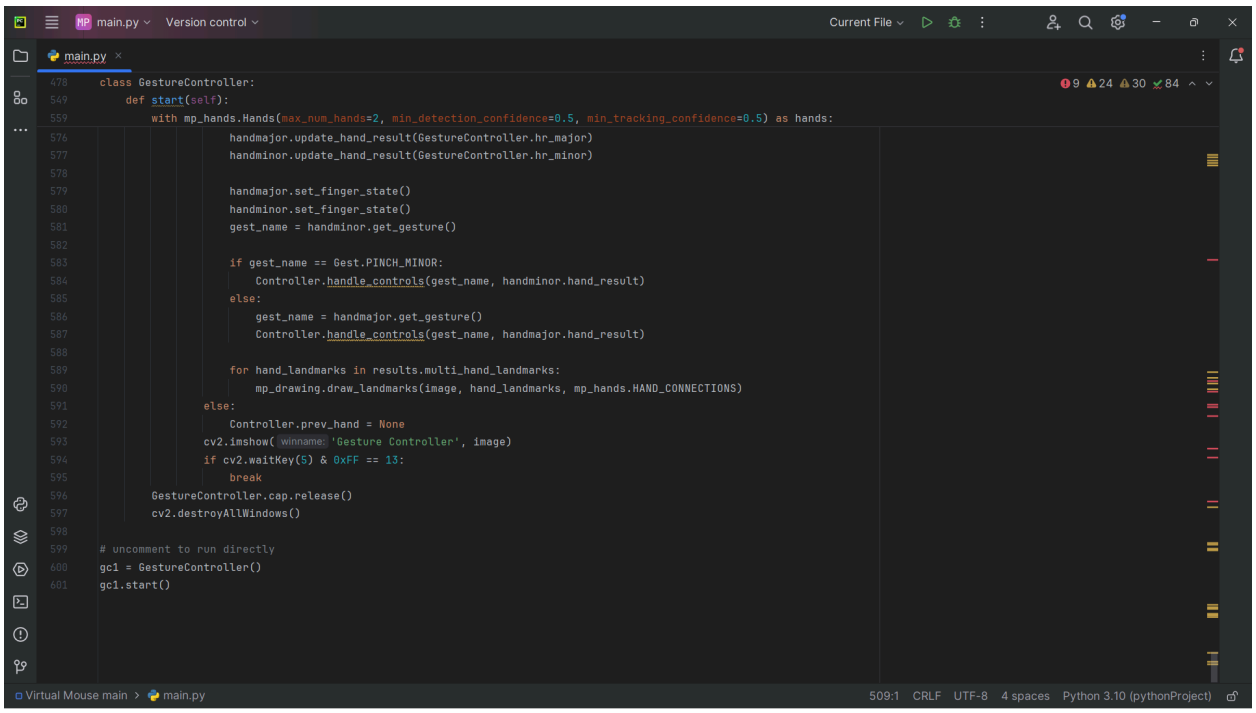
```python
class GestureController:
    """
    gc_mode = 0
    cap = None
    CAM_HEIGHT = None
    CAM_WIDTH = None
    hr_major = None  # Right Hand by default
    hr_minor = None  # Left hand by default
    dom_hand = True

    def __init__(self):
        """Initilaizes attributes."""
        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)

    1 usage
    def classify_hands(results):
        """
        sets 'hr_major', 'hr_minor' based on classification(left, right) of
        hand obtained from mediapipe, uses 'dom_hand' to decide major and
        minor hand.
        """
        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
```



```python
class GestureController:
    def classify_hands(results):
            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

    1 usage
    def start(self):
        """
        Entry point of whole programm, caputres video frame and passes, obtains
        landmark from mediapipe and passes it to 'handmajor' and 'handminor' for
        controlling.
        """

        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()
```



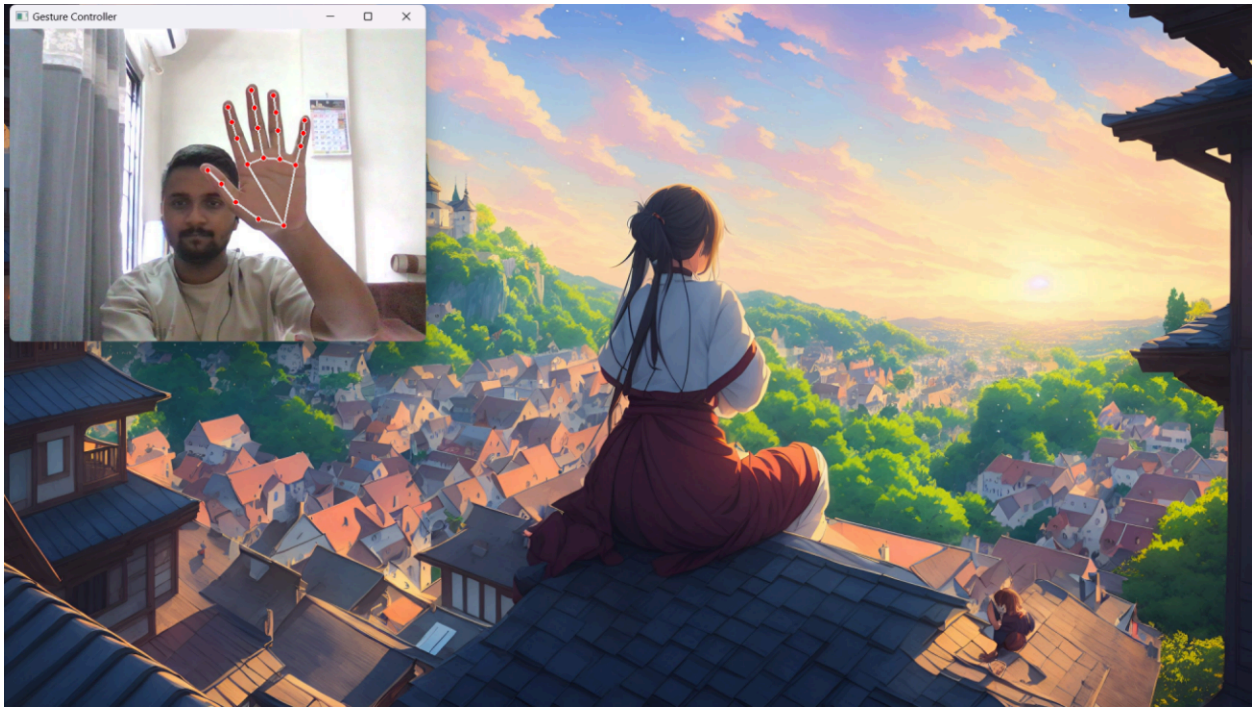
```
class GestureController:
    def start(self):
        handminor = HandRecog(HLabel.MINOR)

        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, flipCode: 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)
```



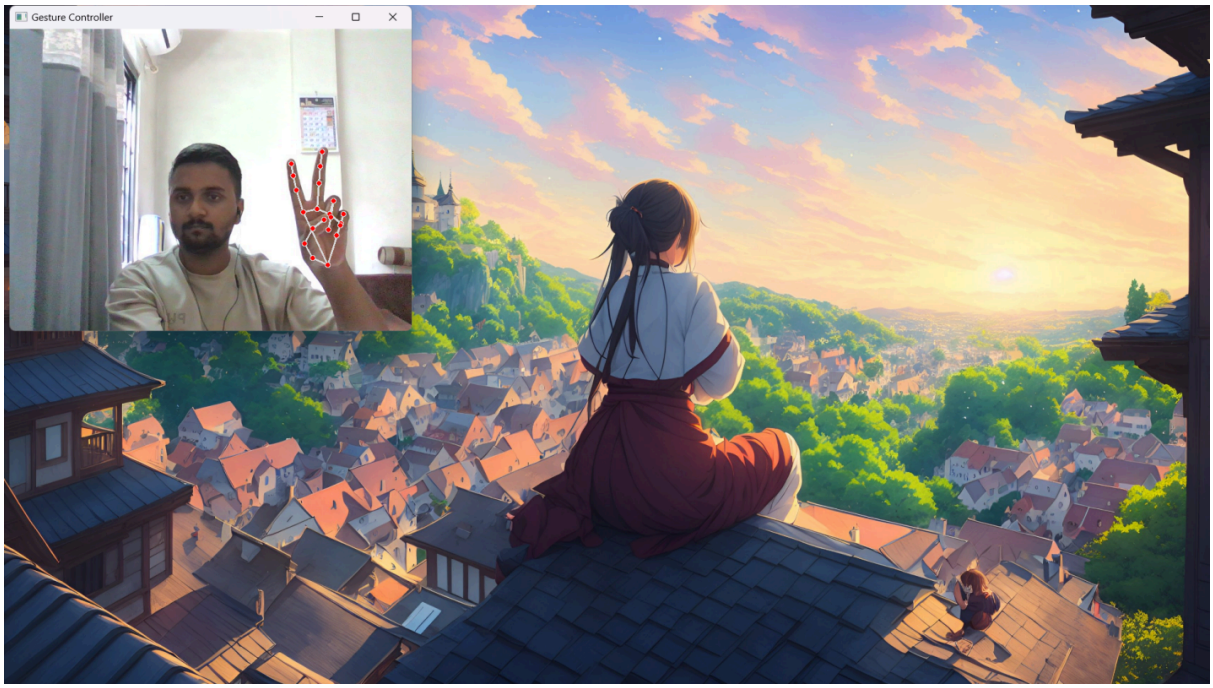
```
class GestureController:
    def start(self):
        with mp_hands.Hands(max_num_hands=2, min_detection_confidence=0.5, min_tracking_confidence=0.5) as hands:
                    handmajor.update_hand_result(GestureController.hr_major)
                    handminor.update_hand_result(GestureController.hr_minor)

                    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( winname: 'Gesture Controller', image)
                if cv2.waitKey(5) & 0xFF == 13:
                    break
        GestureController.cap.release()
        cv2.destroyAllWindows()

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

6.2 Output window:

❖ **Figure 1. Neutral Gesture**

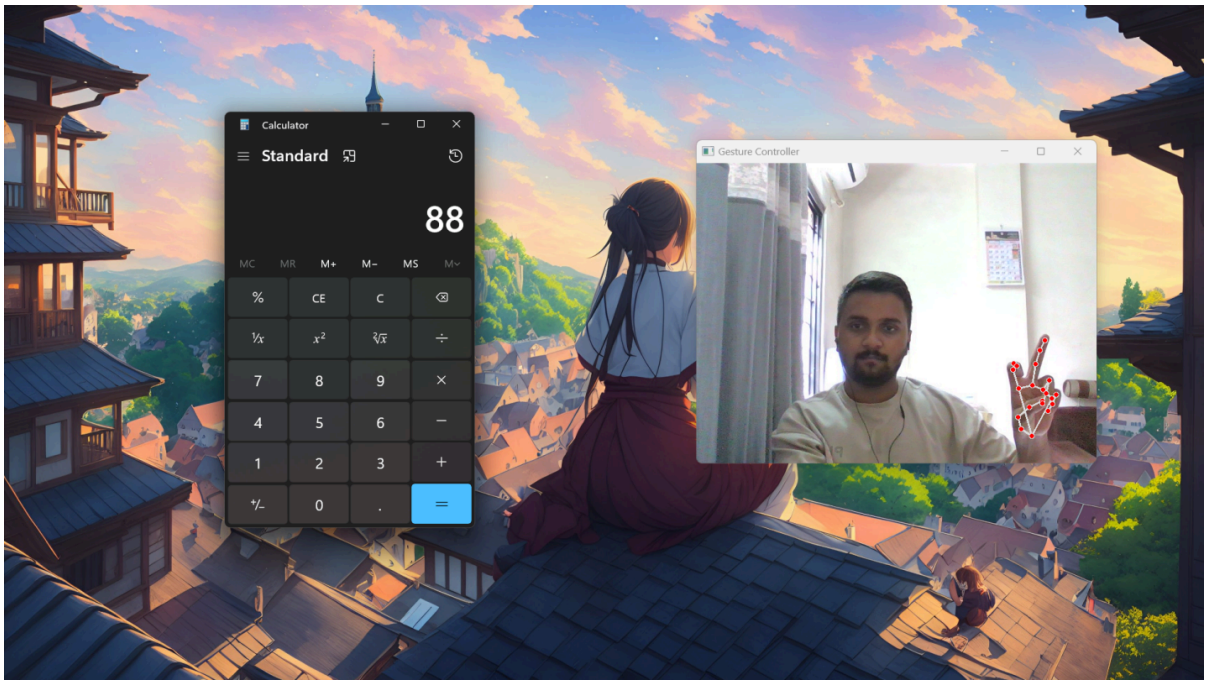


❖ **Figure 2. Move Cursor**

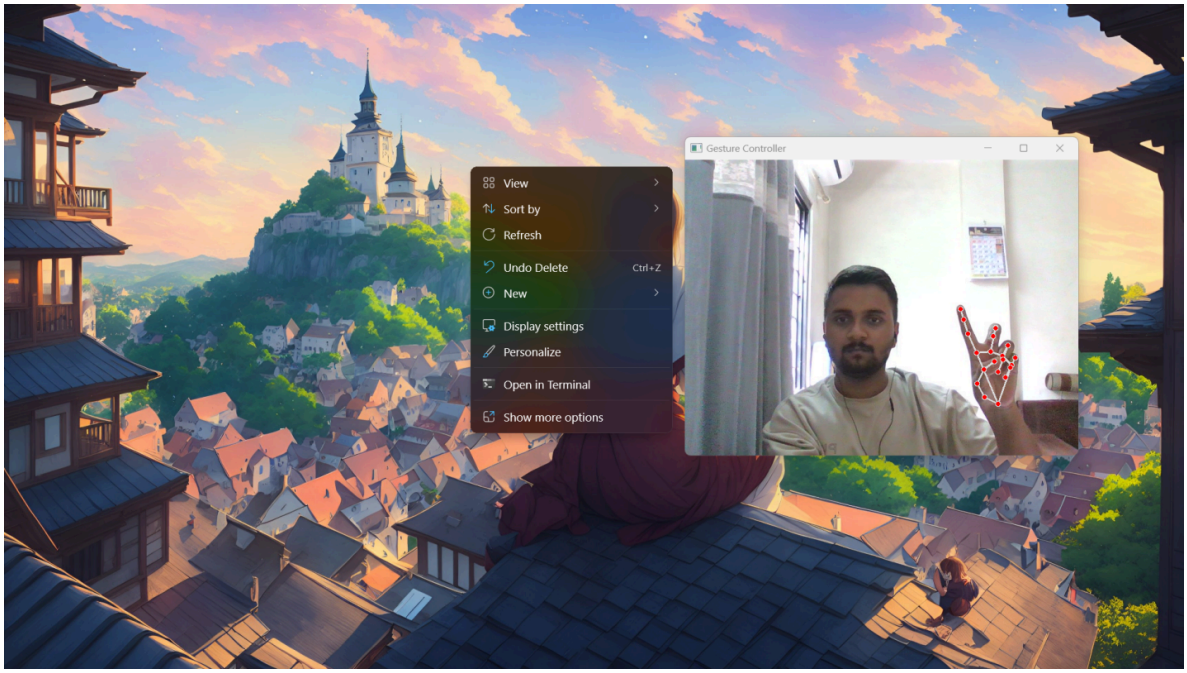


❖

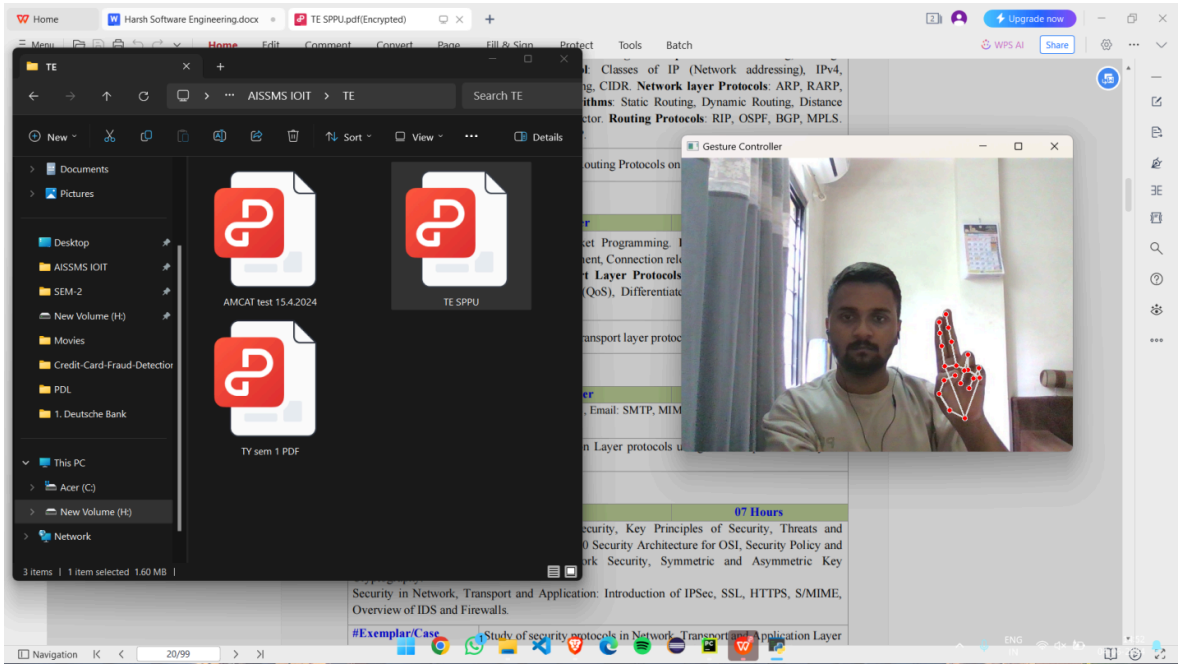
Figure 3. Left Click



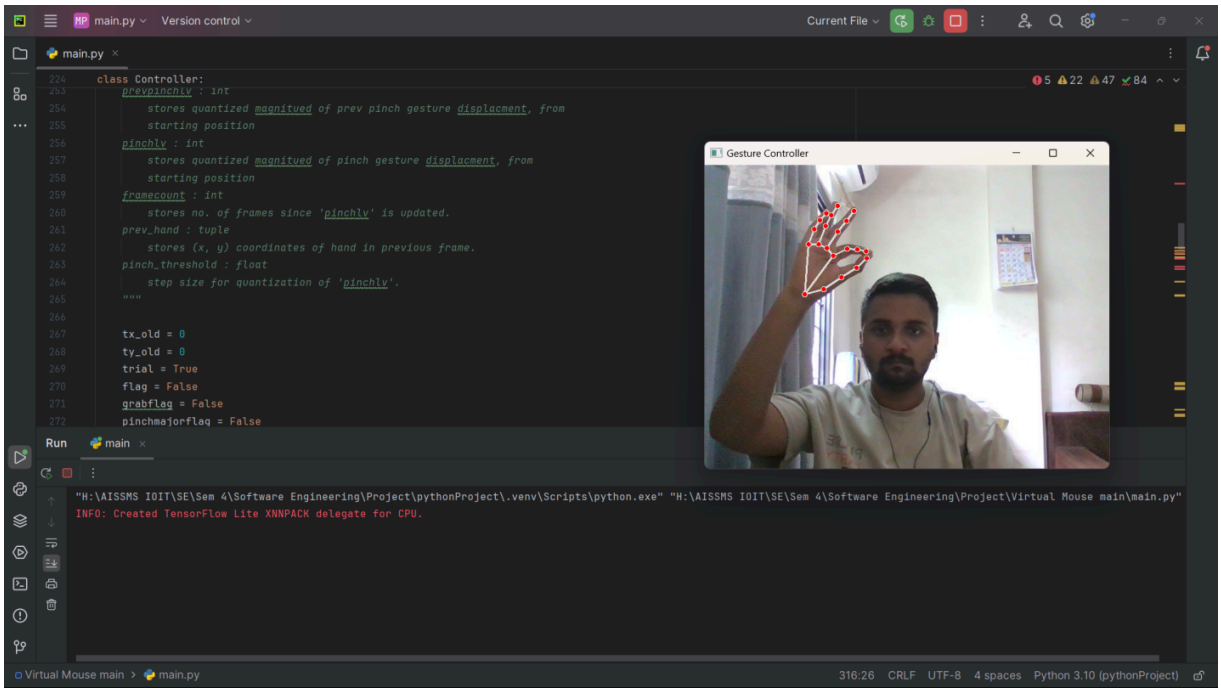
❖ **Figure 4. Right Click**



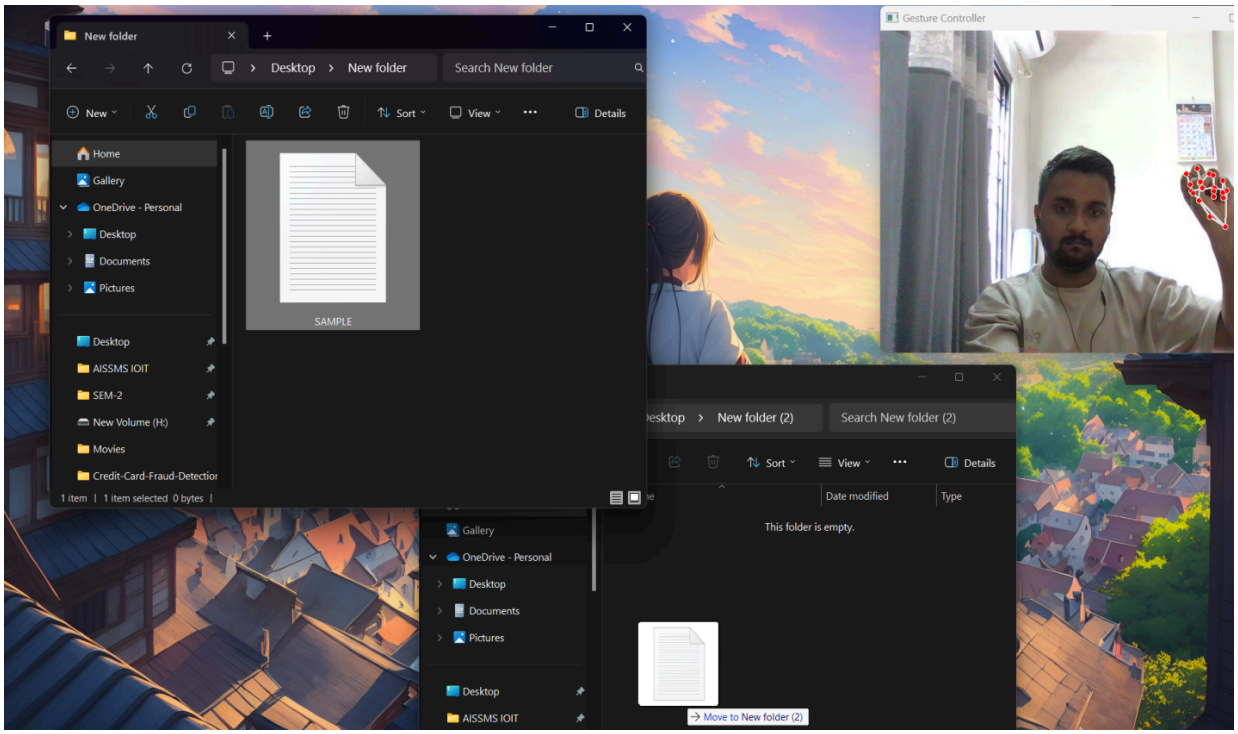
❖ **Figure 5. Double Click**



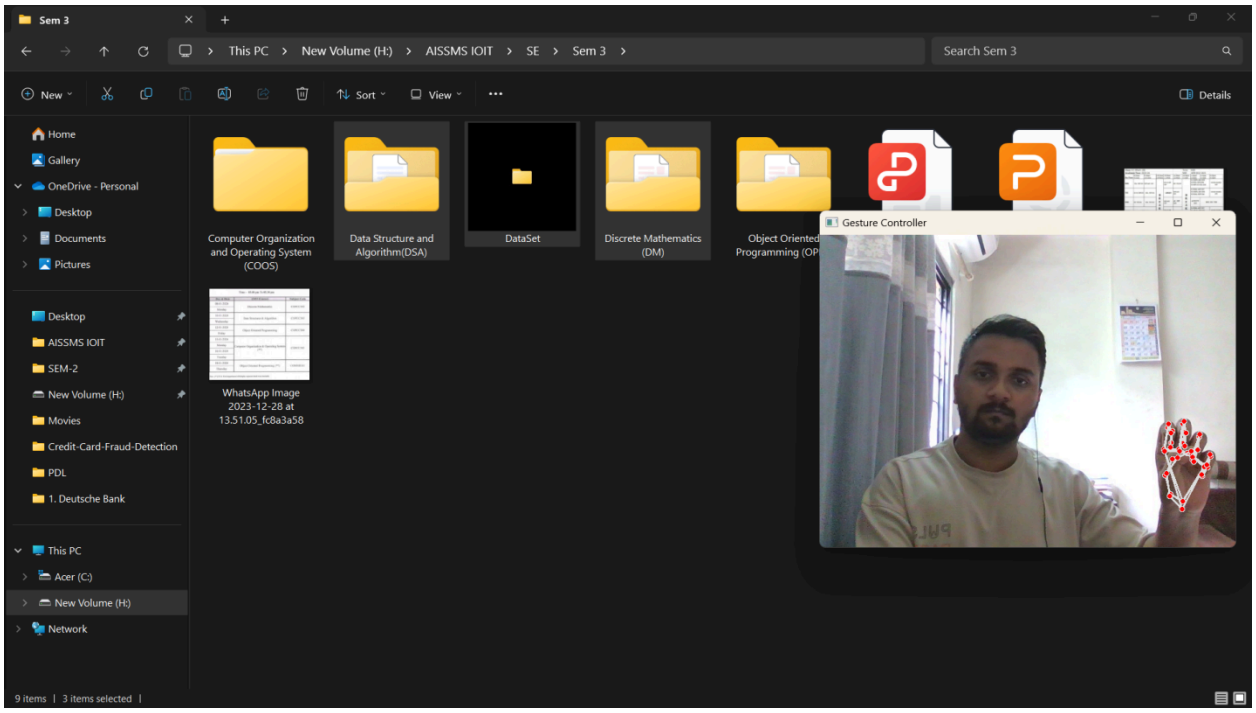
❖ **Figure 6. Scrolling**



❖ **Figure 7. Drag and Drop**



❖ **Figure 8. Multiple Item Selection**



CHAPTER 7 : CONCLUSION AND REFERENCES

7.1 Conclusion

This project successfully implements a real-time hand gesture recognition and control system using Python, Mediapipe, and PyAutoGUI. The system leverages Mediapipe's advanced hand-tracking capabilities to detect and interpret a variety of hand gestures, translating them into computer control functions. Key functionalities include cursor movement, clicking actions, adjusting screen brightness, and changing system volume.

The implementation demonstrates the effective use of gesture recognition technology to create an intuitive and natural user interface. By detecting multiple hand gestures and mitigating noise, the system ensures stable and reliable performance. This stability is crucial for practical applications where precise control is necessary.

The project highlights several important aspects:

1. Gesture Detection and Interpretation: The system can recognize a range of gestures, including fist, palm, thumb, and specific finger combinations. These gestures are mapped to different control actions, enhancing the versatility of the interface.

2. Noise Mitigation: By implementing measures to handle fluctuations due to noise, the system maintains accuracy and responsiveness. This is essential for providing a smooth user experience.

3. Multi-Hand Detection: The ability to distinguish between major and minor hands allows for more complex interactions. For instance, different gestures can be assigned to different hands, enabling simultaneous control of multiple functions.

4. Practical Applications: The system's ability to control brightness and volume, navigate through content, and perform mouse actions showcases its potential for various applications. This can significantly enhance accessibility for users with physical disabilities, improve productivity in professional settings, and offer a novel way of interacting with digital devices.

5. Integration of Technologies: The seamless integration of multiple libraries and tools (Mediapipe for hand tracking, PyAutoGUI for control actions, Pycaw for audio control, and Screen Brightness Control for brightness adjustments) demonstrates the power of combining existing technologies to create new solutions.

Overall, this project illustrates the potential of gesture recognition systems to revolutionize human-computer interaction. It paves the way for more natural and intuitive interfaces, making technology more accessible and user-friendly. Future enhancements could include expanding the range of gestures, improving detection accuracy, and exploring additional control applications. This project serves as a strong foundation for further exploration and development in the field of gesture-based control systems.

7.2 References

· OpenCV (cv2): https://opencv.org/

· MediaPipe (mp): https://ai.google.dev/edge/mediapipe/solutions/guide

· pyautogui: https://pyautogui.readthedocs.io/

· Enum: https://enum-tools.readthedocs.io/en/latest/api/documentation.html

· ctypes: https://docs.python.org/3/library/ctypes.html

· comtypes: https://pythonhosted.org/comtypes/

· pycaw: https://github.com/AndreMiras/pycaw

· google.protobuf: https://developers.google.com/google-ads/api/docs/client-libs/dotnet/working-with-protobuf