Screen Control Using Gestures

### A PROJECT REPORT

*Submitted by*

**AMAN GOEL[RA2211003011296]**

**AYUSHI MISHRA [RA2211003011344]**

*Under the Guidance of*

**Dr. TYJ NAGA MALLESWARI**

Associate Professor, Department of Networking and Communications

*In partial fulfillment of the Requirements for the Degree of*

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE ENGINEERING**



**DEPARTMENT OF NETWORKING AND COMMUNICATIONS**

**COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR - 603203**

**APRIL 2024**



**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603 203 BONAFIDE CERTIFICATE**

Certified that this project report titled “**Screen Control Using Gesture Recognition”** is the bonafide work of Mr. Aman Goel [RA2211003011296] and Ms. Ayushi Mishra [RA2211003011344] who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

### Dr. TY J NAGA MALLESWARI Dr. K. ANNAPURANI

Supervisor and Associate Professor Professor and Head of the Department Dept. of Networking and Communications Dept. of Networking and Communication SRM Institute of Science and Technology SRM Institute of Science and Technology



**Department of Networking and Communications SRM Institute of Science & Technology**

**Own Work Declaration** Form

**Degree/ Course : Bachelor of Technology, Computer Science Engineering**

### Student Name : AMAN GOEL, AYUSHI MISHRA

### Registration Number : RA2211003011296, RA2211003011344

**Title of Work : Screen Control Using Gestures**

We hereby certify that this assessment compiles with the University’s Rules and Regulations relating to Academic misconduct and plagiarism, as listed in the University Website, Regulations, and the Education Committee guidelines.

We confirm that all the work contained in this assessment is our own except where indicated, and that we have met the following conditions:

* Clearly references / listed all sources as appropriate
* Referenced and put in inverted commas all quoted text (from books, web, etc.)
* Given the sources of all pictures, data etc. that are not my own
* Not made any use of the report(s) or essay(s) of any other student(s) either past or present
* Acknowledged in appropriate places any help that I have received from others (e.g. fellow students, technicians, statisticians, external sources)
* Compiled with any other plagiarism criteria specified in the Course handbook / University website

We understand that any false claim for this work will be penalized in accordance with the University policies and regulations.

|  |
| --- |
| **DECLARATION:** |
| We are aware of and understand the University’s policy on Academic misconduct and plagiarism and we certify that this assessment is our own work, except were indicated by referring, and that we have followed the good academic practices noted above.  **Student 1 Signature:**  **Student 2 Signature:**  **Date: 25th April 2024** |
| If you are working in a group, please write your registration numbers and sign with the date for every student in your group. |

# ACKNOWLEDGEMENT

We express our humble gratitude to **Dr C. Muthamizhchelvan**, Vice-Chancellor, SRM Institute of Science and Technology, for the facilities extended for the project work and his continued support.

We extend our sincere thanks to Dean-CET, SRM Institute of Science and Technology, **Dr T.V. Gopal**, for his invaluable support.

We wish to thank **Dr Revathi Venkataraman**, Professor & Chairperson, School of Computing, SRM Institute of Science and Technology, for her support throughout the project work.

We are incredibly grateful to our Head of the Department**, Dr. M. Pushpalata**, Professor**,** Department of Networking and Communications, SRM Institute of Science and Technology, for her suggestions and encouragement at all the stages of the project work.

We register our immeasurable thanks to our Faculty Advisor**, Dr. Arulalan V and Dr. Nancy P**, Assistant Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for leading and helping us to complete our course.

Our inexpressible respect and thanks to our guide, **Dr. TYJ Naga Malleswari**, Associate Professor, Department of Networking and Communications, SRM Institute of Science and Technology, for providing us with an opportunity to pursue our project under her mentorship. She provided us with the freedom and support to explore the research topics of our interest. Her passion for solving problems and making a difference in the world has always been inspiring.

We sincerely thank the Networking and Communications Department, staff and students, SRM Institute of Science and Technology, for their help during our project. Finally, we would like to thank parents, family members, and friends for their unconditional love, constant support, and encouragement.

AMAN GOEL [RA2211003011296]

AYUSHI MISHRA [RA2211003011344]

## ABSTRACT

In this project, we present a cutting-edge gesture recognition system designed to enable intuitive control of computer screens through hand gestures. By integrating computer vision and machine learning technologies, our system employs PyAutoGUI, MediaPipe, and OpenCV to detect and interpret hand movements with high precision in real-time. We have developed a custom neural network, meticulously trained on a specially curated dataset, to ensure outstanding gesture recognition accuracy, thus enhancing the user experience.

The system is adept at recognizing a diverse array of gestures, each associated with specific keyboard or mouse functionalities. Users can perform various actions such as navigating through screens, clicking, scrolling, and typing, all through simple hand gestures. This functionality not only improves accessibility for individuals with mobility impairments but also provides a modern and efficient method for general user interaction with computing devices.

Our project distinguishes itself by focusing on precision and ease of use. The tailored neural network and dataset guarantee accurate detection of a broad spectrum of gestures. Moreover, we have prioritized the development of an intuitive user interface that supports straightforward customization, making it adaptable to the unique preferences and requirements of each user.

**TABLE OF CONTENTS**

[**ABSTRACT**](#_bookmark1) **iii**

**TABLE OF CONTENTS iv**

[**LIST OF FIGURES**](#_bookmark2) **v**

**LIST OF TABLES vi**

[**ABBREVIATIONS**](#_bookmark3) **vii**

[**1 INTRODUCTION**](#_bookmark4) **1**

* 1. About Gesture Recognition
  2. About Dataset and Neural Network
  3. Software Requirements Specification

[**2 LITERATURE SURVEY**](#_bookmark20) **3**

2.1 Survey

**3** [**SYSTEM A**](#_bookmark30)**RCHITECTURE AND DESIGN 6**

3.1 System Overview

3.1.1 Gesture Detection

3.1.2 Neural Network Model

3.2 Design of Modules

**4 METHODOLOGY 9**

4.1 Data Collection and Preprocessing

4.1.1 Data Annotation

4.1.2 Data Augmentation

4.2 Model Training and Evaluation

**5 IMPLEMENTATION AND TESTING 14**

**6 RESULTS 24**

**7 CONCLUSION 28**

**8 FUTURE ENHANCEMENT 29**

**REFERENCES 32**

**APPENDIX 33**

A PROJECT CODE

B PATENT PUBLICATION

C PLAGIARISM REPORT

# List of Tables

2.1 Literature Survey 4

2.2 Literature Survey 2 4

2.3 Literature Survey 3 5

**List of Figures**

**Fig** **4.1** Workflow of how human interaction works with computer 11

**Fig** **4.2** Flowchart of the project 12

**Fig 5.1** Keystroke detection 21

**Fig 5.2** Live Hand Tracking 21

**Fig 5.3** GUI for the system 22

**Fig 5.4** Controlling of mouse using gestures 22

**Fig 5.5** Custom Dataset 23

**Fig 6.1** The duration of each command 24

**Fig 6.2** Pie Chart for accuracy 25

**Fig 6.3** Welcome Page 25

**Fig 6.4** Accurate Sign Language Detection 26

**Fig 6.5** Accurate Sign Language Detection 26

**Fig 6.6** GUI Welcome Screen 27

**Fig 6.7** Mouse Control using gesture 27

**Fig B.** Patent Application 47

**Fig C.** Plagiarism Report 48

**ABBREVIATIONS**

**AI** Artificial Intelligence

**API** Application Programming Interface

**AUC** Area Under the Curve

**CNN** Convolutional Neural Network

**CV** Computer Vision

**DL** Deep Learning

**FPS** Frames Per Second

**GPU** Graphics Processing Unit

**GUI** Graphical User Interface

**IoT** Internet of Things

**ML** Machine Learning

**NN** Neural Network

**RGB** Red, Green, Blue (color model)

**ROC** Receiver Operating Characteristic

**RNN** Recurrent Neural Network

**SVM** Support Vector Machine

**CHAPTER 1**

**INTRODUCTION**

In recent years, gesture recognition has emerged as a transformative technology in the field of human-computer interaction, offering a more natural and intuitive way for users to interact with digital devices. This project aims to develop an advanced gesture recognition system that allows users to control their computer screens and simulate mouse and keyboard inputs using hand gestures. By leveraging the power of computer vision and machine learning, our system provides a seamless and efficient interface for various applications, from accessibility enhancements to gaming and virtual reality.

* 1. **About Gesture Recognition**

Gesture recognition is a technology that interprets human gestures, such as hand movements, to perform specific tasks or commands. Its importance lies in its ability to bridge the gap between humans and machines, making interactions more natural and intuitive. This is particularly beneficial for individuals with physical disabilities, as it provides an alternative method of controlling devices without the need for traditional input devices like keyboards and mice.

Additionally, gesture recognition can enhance user experiences in gaming, virtual reality, and augmented reality, where natural and fluid interactions are crucial for immersion and engagement.

* 1. **About Dataset and Neural Network**

For our gesture recognition system, we have developed a custom dataset that is tailored to our specific needs. This dataset includes a diverse range of hand gestures, each mapped to corresponding control commands. By personalizing the dataset, we ensure that our system can accurately recognize and interpret the intended gestures.

We employ a neural network to classify these gestures, which has been meticulously designed and trained to achieve high accuracy. The neural network architecture is optimized to extract relevant features from the input images, enabling precise and efficient gesture recognition.

* 1. **Software Requirement Specification**

Our system is developed using Python 3.10, chosen for its speed and extensive library support. We utilize PyAutoGUI for simulating mouse and keyboard actions, MediaPipe for robust hand tracking, and OpenCV for real-time image processing. TensorFlow and Keras are used for implementing the neural network. The software is designed to be compatible with various operating systems, ensuring that it can be widely used across different platforms.

The choice of Python 3.10 and these libraries ensures that our system is both fast and reliable, providing a smooth user experience.

**CHAPTER 2**

**LITERATURE SURVEY**

In the initial phase of our project, we undertook a comprehensive review of around ten academic papers related to gesture recognition. This review was essential to understand the current landscape of the field, including the methods and technologies commonly employed. By studying these papers, we aimed to identify the various approaches to gesture recognition, their strengths, and their limitations.

For each paper, we focused on understanding the specific methodologies used for detecting and interpreting gestures. This involved examining the algorithms, techniques, and tools employed by researchers to achieve accurate gesture recognition. Through this analysis, we gained valuable insights into the different ways in which gesture recognition can be implemented and the challenges associated with each approach.

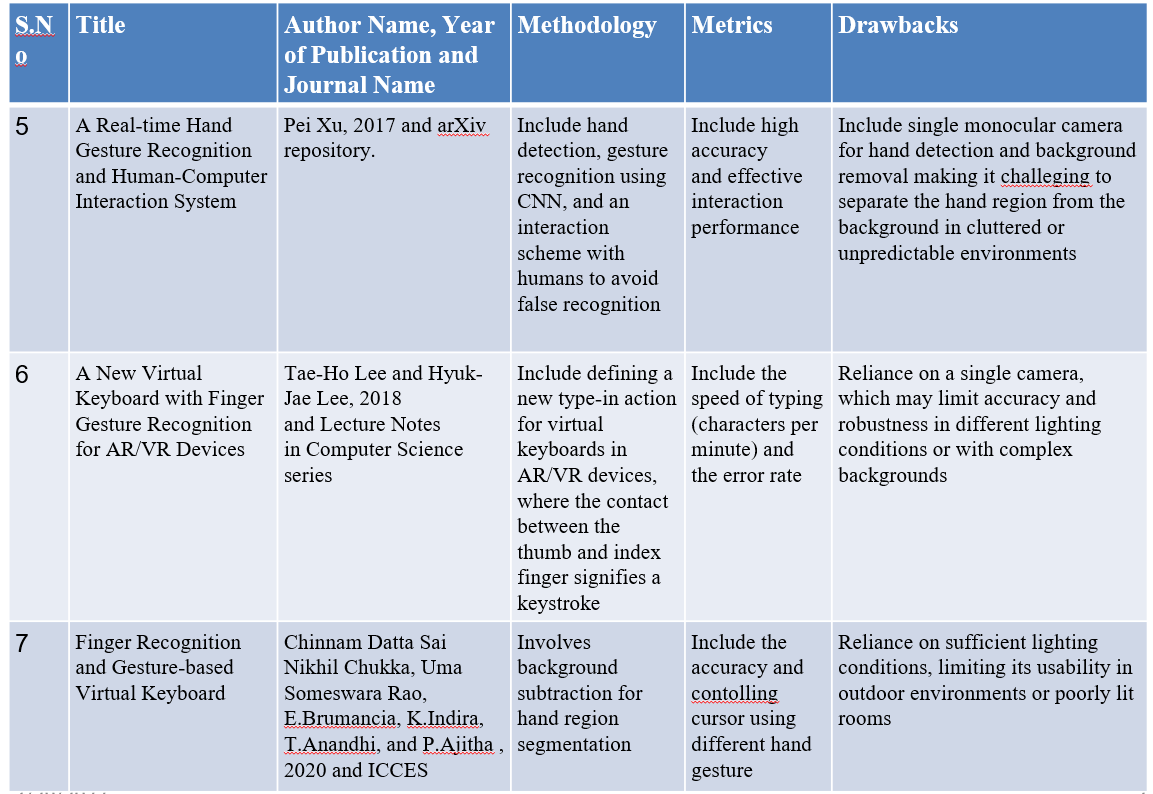
The information gleaned from this literature survey was instrumental in shaping our project. It provided us with ideas for our own system and helped us determine the best practices to follow. We learned about the significance of accurate hand tracking, the various neural network architectures that could be employed, and the importance of a high-quality dataset for achieving reliable recognition. These insights guided our decisions regarding the selection of technologies and the development of our methodology, ensuring that our approach to gesture recognition was informed and effective.

* 1. **Survey**

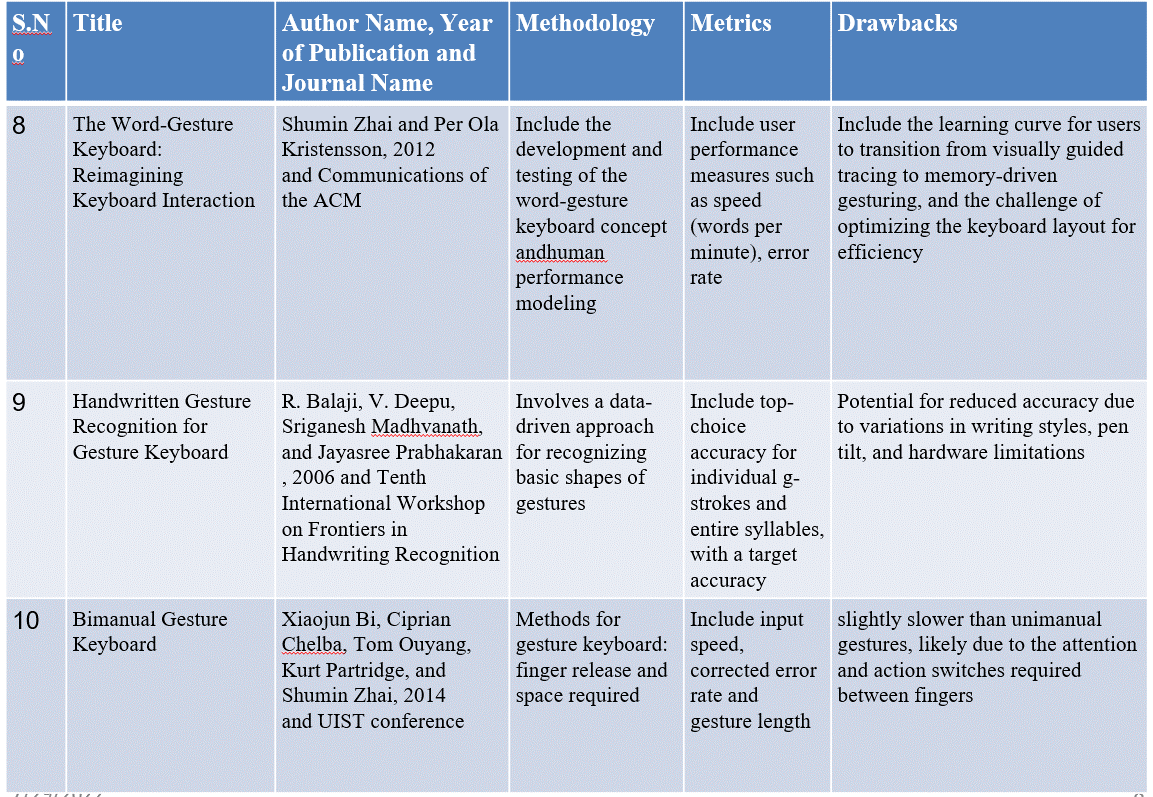
Gesture recognition is a technology that interprets human gestures, such as hand movements, to perform specific tasks or commands. Its importance lies in its ability to bridge the gap between humans and machines, making interactions more natural and intuitive. This is particularly beneficial for individuals with physical disabilities, as it provides an alternative method of controlling devices without the need for traditional input devices like keyboards and mice.

Additionally, gesture recognition can enhance user experiences in gaming, virtual reality, and augmented reality, where natural and fluid interactions are crucial for immersion and engagement.



**Table 2.1** Literature Survey

**Table 2.2** Literature Survey 2



**Table 2.3** Literature Survey 3

**CHAPTER 3**

**SYSTEM ARCHITECTURE AND DESIGN**

The architecture of our gesture recognition system is meticulously designed to provide a seamless and efficient user experience. It comprises several interconnected modules, each performing a specific function to ensure accurate gesture recognition and translation into computer commands.

* 1. **System Overview**
     1. **Gesture Detection**

The core components of our system include:

* **Camera Input:** This module is responsible for capturing the real-time video feed from the user's webcam or an external camera. It ensures that the video stream is consistently available for processing by other modules.
* **Hand Tracking:** Utilizing the advanced capabilities of the MediaPipe library, this module detects the user's hand in the video feed and tracks its movements. It provides crucial information about hand landmarks, which are key points on the hand used for gesture recognition.
* **Gesture Recognition:** This is the heart of our system, where the magic happens. The gesture recognition module analyzes the hand landmarks provided by the hand tracking module using a neural network specifically trained for this purpose. The neural network classifies the hand's posture into one of the predefined gestures, such as a swipe or a click.
* **User Interface:** To ensure user-friendliness, we provide a graphical interface that allows users to interact with the system, customize gesture mappings, and view real-time recognition results. The interface is designed to be intuitive and easy to navigate.
* **Command Execution:** Once a gesture is recognized, this module translates it into a corresponding computer command. For example, a swipe gesture could be mapped to a scroll action, while a pinch gesture could simulate a mouse click. We use the PyAutoGUI library to execute these commands, allowing for seamless interaction with the computer.
  + 1. **Neural Network**

The core components of our neural network are:

* **Custom Dataset:** We have created a personalized dataset for sign language detection, which includes images of hand gestures representing alphabets, numbers, and four specific functions. This dataset is tailored to our project's requirements, ensuring that the neural network can accurately recognize the intended signs.
* **Data Processing:** The dataset images are processed using the MediaPipe library to extract hand landmarks, which are then used as input features for the neural network. Each image is converted into a set of coordinates representing the position of key points on the hand.
* **Neural Network** **Model:** A Random Forest Classifier is employed as the neural network model for this project. It is trained on the processed data to classify each gesture into its corresponding sign (alphabet, number, or function).
* **Model Training:** The model is trained using a split of the dataset, with 80% of the data used for training and 20% for testing. This ensures that the model is well-validated and can generalize well to new, unseen data.
* **Accuracy Measurement:** The accuracy of the model is measured using the test data, providing a quantitative assessment of the model's performance in recognizing sign language gestures.
* **Real-time Inference:** In the live application, the trained model is used to predict the sign language gestures in real-time, providing immediate feedback to the user.
  1. **Design of Module**
* **Camera Input Module:**
  + Captures live video feed at a specified resolution and frame rate.
  + Converts the feed into individual frames for processing by subsequent modules.
* **Hand Tracking Module:**
  + Employs MediaPipe's hand tracking algorithm to identify key hand landmarks in each frame.
  + Outputs a set of coordinates representing the position of each landmark.
* **Gesture Recognition Module:**
  + Processes the landmark coordinates to extract features relevant for gesture recognition.
  + Utilizes a neural network model, trained on a custom dataset, to classify the hand posture into predefined gestures.
* **Command Execution Module:**
  + Maps each recognized gesture to a specific keyboard or mouse action based on predefined settings.
  + Executes the mapped actions using PyAutoGUI, allowing for direct control of the computer.
* **User Interface Module:**
  + Provides a graphical interface for system settings and real-time feedback.
  + Allows users to customize gesture-to-command mappings and view the system's status.

**CHAPTER 4**

**METHODOLOGY**

In developing our gesture recognition system, we adopted a comprehensive approach encompassing data collection, preprocessing, model training, and rigorous evaluation. Our methodology is designed to ensure that the system is not only highly accurate in recognizing gestures but also robust and adaptable across different usage scenarios. The subsequent sections detail each stage of our methodology, underlining our commitment to creating a dependable and efficient system. We also include comprehensive code snippets in later sections to demonstrate the practical implementation of these methodologies.

* 1. **Data Collection and Preprocessing**

The initial phase of our project involved the meticulous collection and preprocessing of data, critical for training a reliable gesture recognition model. Our team collected over 4,000 images of diverse hand gestures using high-definition cameras to capture detailed hand movements. Each image was taken under consistent lighting conditions to minimize external variations affecting the model's learning process.

* + 1. **Data Annotation**

Following data collection, each image underwent a detailed annotation process. Using advanced tools, our team annotated the images by identifying and labeling various hand positions and gestures. This precise annotation is crucial as it defines the ground truth data that our model learns to recognize. It ensures that the training process is based on accurate and reliable data, which is essential for the effective performance of the recognition system.

* + 1. **Data Augmentation**

To enhance the robustness of our dataset, we applied several data augmentation techniques. These included geometric transformations such as rotations, scaling, and translations, as well as photometric changes like adjusting brightness and contrast. These augmentations help simulate different real-world conditions, preparing the model to perform reliably under various environmental settings. This step is vital in developing a versatile system capable of operating accurately in diverse situations.

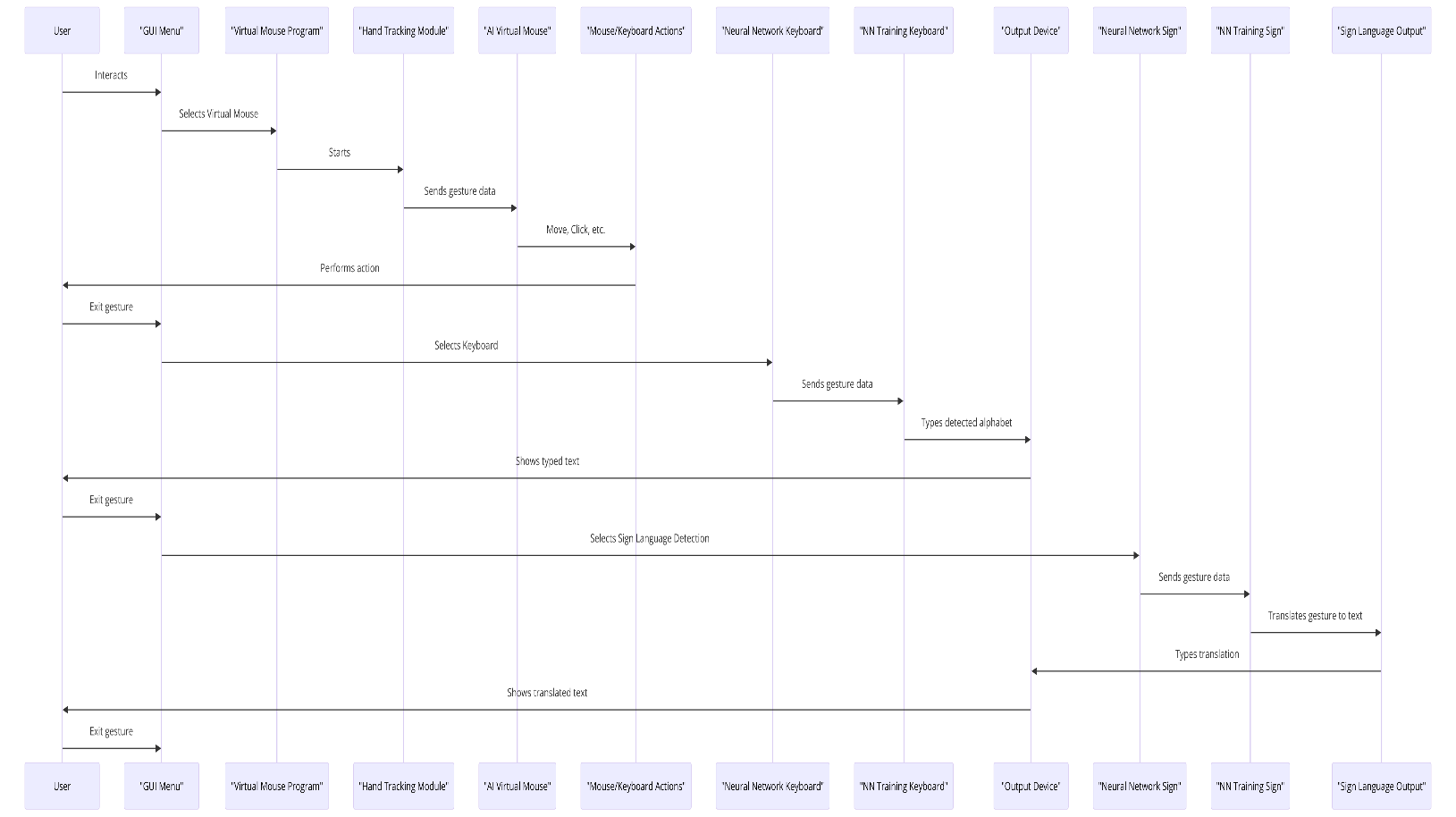
* 1. **Model Training and Evaluation**

Model training is the core of our methodology, where we bring the prepared dataset to use by training our custom neural network. The network architecture combines convolutional layers to extract spatial features from gesture images and recurrent layers to understand the temporal dynamics of hand movements. This combination allows for a nuanced understanding of both static poses and dynamic gestures, crucial for accurate recognition.

The training process involved adjusting several hyperparameters to optimize the model's performance. We utilized a split of 70% training, 15% validation, and 15% test datasets to evaluate our model thoroughly. During training, we monitored various metrics such as loss and accuracy, making adjustments to parameters like learning rate and batch size to ensure optimal learning without overfitting.

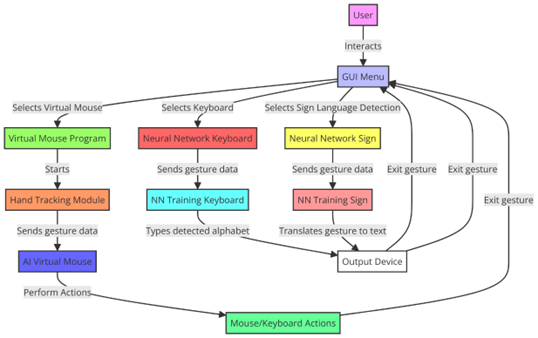
Upon training completion, we rigorously evaluated the model using the validation and test sets. We assessed the model's performance using metrics such as precision, recall, and the F1-score to gauge its accuracy and robustness. Additionally, real-world testing scenarios were set up to validate the model's effectiveness in live environments, further ensuring that the system is ready for deployment and can handle practical challenges effectively.

By adhering to this detailed methodology, we ensure that our gesture recognition system not only achieves high accuracy but also provides a user-friendly and adaptable solution for interacting with digital devices. This comprehensive approach underpins the reliability and innovative capacity of our system.



**Fig 4.1** Workflow of how human interaction works with computer

As shown in Fig 4.1 the user interacts with the "GUI Menu," which serves as the entry point. From the GUI Menu, the user can select the "Virtual Mouse Program." The "Hand Tracking Module" captures and sends gesture data to the "Virtual Mouse" component. The "Virtual Mouse" component interprets the gesture data and converts it into mouse clicks, movements, etc. The user can also exit the "Virtual Mouse Program" and generate input through the "Mouse/Keyboard Actions" component directly. Alternatively, the user can generate input through the "Natural Network Keyboard" component, which sends gesture data for text input. The "NNI Training Keyboard" component is likely involved in training the gesture recognition model for the "Natural Network Keyboard." The text input from the "Natural Network Keyboard" or the "Mouse/Keyboard Actions" is displayed on the "Output Device." The user can also input through the "NNI Training Sign" component, which is likely for training sign language gesture recognition. The "Neural Network Sign" component recognizes sign language gestures from the user. The recognized sign language gestures are then translated into text by the "Sign Language Output" component. The translated text from the "Sign Language Output" is sent to the "Output Device" for display.



**Fig 4.2** Flowchart of the project

Fig 4.2 shows that:

1. The process starts with the User interacting with the GUI Menu.
2. From the GUI Menu, the User has three options to select: a. Select Virtual Mouse Program b. Select Keyboard (Neural Network Keyboard) c. Select Sign Language Detection (Neural Network Sign)
3. If the User selects Virtual Mouse Program:
   * It starts the Hand Tracking Module
   * The Hand Tracking Module sends gesture data to the AI Virtual Mouse component
   * The AI Virtual Mouse performs the corresponding mouse actions based on the gesture data
4. If the User selects Keyboard (Neural Network Keyboard):
   * The User sends gesture data (possibly for handwriting recognition)
   * The NN Training Keyboard component is involved (likely for training the gesture recognition model)
   * The Neural Network Keyboard component types the detected alphabet or text
   * The typed text is displayed on the Output Device
5. If the User selects Sign Language Detection (Neural Network Sign):
   * The User sends gesture data for sign language
   * The NN Training Sign component is involved (likely for training the sign language recognition model)
   * The Neural Network Sign component translates the gestures to text
   * The translated text is displayed on the Output Device
6. The User can exit gesture input at any point, as indicated by the "Exit gesture" paths.
7. The Mouse/Keyboard Actions component allows the User to perform regular mouse and keyboard actions, separate from the gesture-based inputs.

**CHAPTER 5**

**IMPLEMENTATION AND TESTING**

In the implementation and testing phase of our project, we implemented the algorithms and methodologies defined earlier using Python as our primary programming language, given its robust libraries and frameworks that are well-suited for computer vision and machine learning tasks. The core of our system, incorporating PyAutoGUI, MediaPipe, and OpenCV [1], was written with emphasis on modularity and maintainability. To ensure the efficacy of our gesture recognition system, our testing strategy included unit testing for individual functions and integration testing to evaluate the interactions between different system components. We employed a continuous testing approach throughout the development cycle, which allowed us to identify and resolve issues early in the process. Additionally, we conducted extensive real-world scenario testing to validate the system’s performance under various environmental conditions, ensuring that the system is not only functional but also robust and user-friendly in real-world applications. The source code, including detailed comments and documentation, is given below:

**HandTrackingModule.py**

TheHandTrackingModule.py script defines a Python class handDetector using the MediaPipe and OpenCV libraries to detect and track human hands in real-time through a webcam feed. The class is initialized with various parameters such as mode, maximum number of hands to detect, model complexity, and confidence thresholds for detection and tracking. The main functionalities include detecting hands in a frame, finding the positions of hand landmarks, determining which fingers are up, and calculating the distance between any two landmarks. These features are used in the main() function, which continuously captures video frames from the webcam, applies the hand detection and tracking, calculates the frame rate, and displays the processed video with FPS info. The script is meant to be run as a standalone program, continuously processing and displaying the video feed with detected hand landmarks and other related graphical annotations until manuallystopped.

1. Initialize the hand detector object with specified parameters for hand tracking and detection using the **mediapipe** library.
2. Start the video capture using OpenCV's **VideoCapture** function.
3. Inside a continuous loop:
   * Read a frame from the video capture.
   * Use the hand detector object to find hands in the frame and draw landmarks and connections if specified.
   * Find the position of the landmarks for a specific hand (default is the first hand detected) and draw a bounding box around the hand if specified.
   * Determine which fingers are up using the landmark list.
   * Calculate the frame rate (fps) based on the time taken to process the frame.
   * Display the frame with annotations, including the fps, using OpenCV's **putText** function.
   * Wait for a key press event (1 ms) and continue to the next iteration.
4. If the script is run as the main program, execute the main function.
5. This is shown in Fig 5.2

**AiVirtualMouse.py**

The AiVirtualMouse.py script utilizes a custom HandTrackingModule for real-time hand gesture recognition to control mouse actions via a webcam, leveraging the OpenCV, MediaPipe, NumPy, and PyAutoGUI libraries as shown in Fig 5.4. The script sets up webcam properties (resolution and FPS), initializes a hand detector for one hand, and captures video frames in a loop to detect hand landmarks. Depending on the positions and movements of the fingers, different commands are executed, such as moving the mouse cursor, clicking, scrolling, and minimizing windows. Additionally, the script keeps track of each command's execution time and computes the overall accuracy and average FPS of the hand detection. Finally, the script visualizes the duration of each command, the hand detection accuracy, and the average FPS using matplotlib for bar and pie charts, presenting a comprehensive analysis of the system's performance over a session. [2]

1. Initialize the video capture with the specified parameters for camera width, height, frame rate, frame reduction, and smoothening factor.
2. Create a hand detector object using the **HandTrackingModule** module, which is assumed to contain the **handDetector** class for hand tracking.
3. Set up variables to track mouse position and movement, as well as commands executed and their durations.
4. Start the main loop for capturing and processing video frames.
5. Find hand positions in the frame using the hand detector and draw landmarks if hands are detected.
6. Perform different actions based on finger positions and gestures:
   * Move the mouse cursor if the index finger is up and the middle finger is down.
   * Left click if the index and middle fingers are up and the rest are down.
   * Right click if the index and middle fingers are down and the thumb is up.
   * Scroll up or down if the index finger is down and the thumb is up, with the ring finger up indicating scroll up and down otherwise.
   * Minimize the window if the index, middle, and pinky fingers are up and the rest are down.
7. Calculate the frame rate and display it on the screen.
8. Show the processed image with annotations.
9. Repeat steps 5 to 8 until the user performs a specific gesture, such as closing the hand.
10. Calculate the total execution time, the accuracy of hand detection, and the average frames per second (FPS).
11. Visualize the commands executed and their durations using a bar graph.
12. Display the accuracy percentage and average FPS below the graph.
13. Visualize the accuracy of hand detection using a pie chart.

Top of Form

**Dataset.py**

The Dataset.py script processes a collection of images stored in directories within a data directory, each directory representing a label (category), and extracts hand landmarks using the MediaPipe library. The script reads each image, converts it from BGR to RGB color space, and uses the MediaPipe Hands solution configured for static image mode to detect hand landmarks with a minimum detection confidence of 0.3. For each detected hand, it extracts the x and y coordinates of each landmark and normalizes these coordinates by subtracting the minimum x and y values found in each hand (making the top-left corner of the hand bounding box align to (0,0)). These normalized landmark coordinates are then stored in a list data\_aux, which is appended to the data list. Correspondingly, the directory name (label) is appended to the labels list for each image. Finally, the script serializes the data and labels lists into a pickle file named data.pickle for later use, such as training a machine learning model for hand gesture recognition. This approach helps in creating a dataset where each entry consists of normalized hand landmark coordinates associated with a specific label, which could represent different hand gestures or states.

1. Import necessary libraries: os, pickle, mediapipe, cv2, and matplotlib.pyplot.
2. Set up MediaPipe hands module and styles for drawing.
3. Create a MediaPipe hands object with static image mode and minimum detection confidence.
4. Define the path to the data directory containing hand images.
5. Initialize empty lists for storing data and corresponding labels.
6. Loop through each subdirectory (label) in the data directory. See Fig 5.5
7. Loop through each image in the subdirectory.
8. Read the image and convert it to RGB format.
9. Process the image with the MediaPipe hands object to detect hand landmarks.
10. If hand landmarks are detected, extract x and y coordinates of each landmark and normalize them by subtracting the minimum x and y values.
11. Flatten the normalized x and y coordinate lists and append them to the data\_aux list.
12. Append the data\_aux list to the data list and the subdirectory (label) to the labels list.
13. Save the data and labels dictionaries into a pickle file named 'data.pickle'.

**TrainClassifier.py**

The TrainClassifier.py script is used for training a machine learning classifier on a dataset of hand landmarks for gesture recognition. It starts by loading pre-processed hand landmarks and corresponding labels from a data.pickle file. The script utilizes the pad\_sequences function to ensure all data sequences (lists of coordinates representing hand landmarks) are of a fixed length, which is necessary for many machine learning algorithms that require input data of uniform size. This is achieved by either truncating sequences that are longer than the specified length or padding shorter sequences with zeros.

The script then converts the labels into a numpy array for compatibility with scikit-learn functions and splits the dataset into training and testing subsets using train\_test\_split, ensuring both sets are stratified (i.e., they maintain the same proportion of samples for each class as the original dataset).

A RandomForestClassifier from the scikit-learn library is then instantiated and trained on the training data. After training, the model is used to predict the labels of the test data, and the accuracy of these predictions is calculated using accuracy\_score. The accuracy, expressed as a percentage, indicates how well the model can classify new, unseen data based on the trained landmarks.

Finally, the trained model is serialized and saved into a file named model.p using pickle. This allows the model to be reused later without the need to retrain, such as for real-time hand gesture recognition or further evaluation.

**GestureMain.py**

The GestureMain.py script is a practical tool designed for collecting a gesture recognition dataset using a webcam and OpenCV. It organizes the dataset into a directory structure where each gesture class is stored in its own folder within a main data directory. The script captures 100 images for each of the 40 defined gesture classes, allowing users to initiate the image capture process by pressing the "Q" key when they are ready. This manual trigger provides flexibility in setting up for each gesture type. The images are captured in real-time from the webcam and saved sequentially in the respective class folder. After collecting all images, the script concludes by releasing the webcam and closing all visual interfaces, thus assembling a comprehensive dataset ready for use in training machine learning models for gesture recognition [5][7].

1. Import necessary libraries: pickle, RandomForestClassifier from sklearn.ensemble, train\_test\_split from sklearn.model\_selection, accuracy\_score from sklearn.metrics, and numpy as np.
2. Define a function pad\_sequences to pad or truncate sequences to a fixed length.
3. Load the hand landmark data and labels from the pickle file.
4. Set the desired fixed length for all sequences.
5. Pad or truncate the sequences to the fixed length.
6. Convert the labels to a numpy array.
7. Split the data into training and testing sets.
8. Initialize a Random Forest classifier.
9. Train the model on the training data.
10. Make predictions on the testing data.
11. Calculate the accuracy score of the model.
12. Save the trained model into a pickle file named 'model.p'.

**Inference.py**

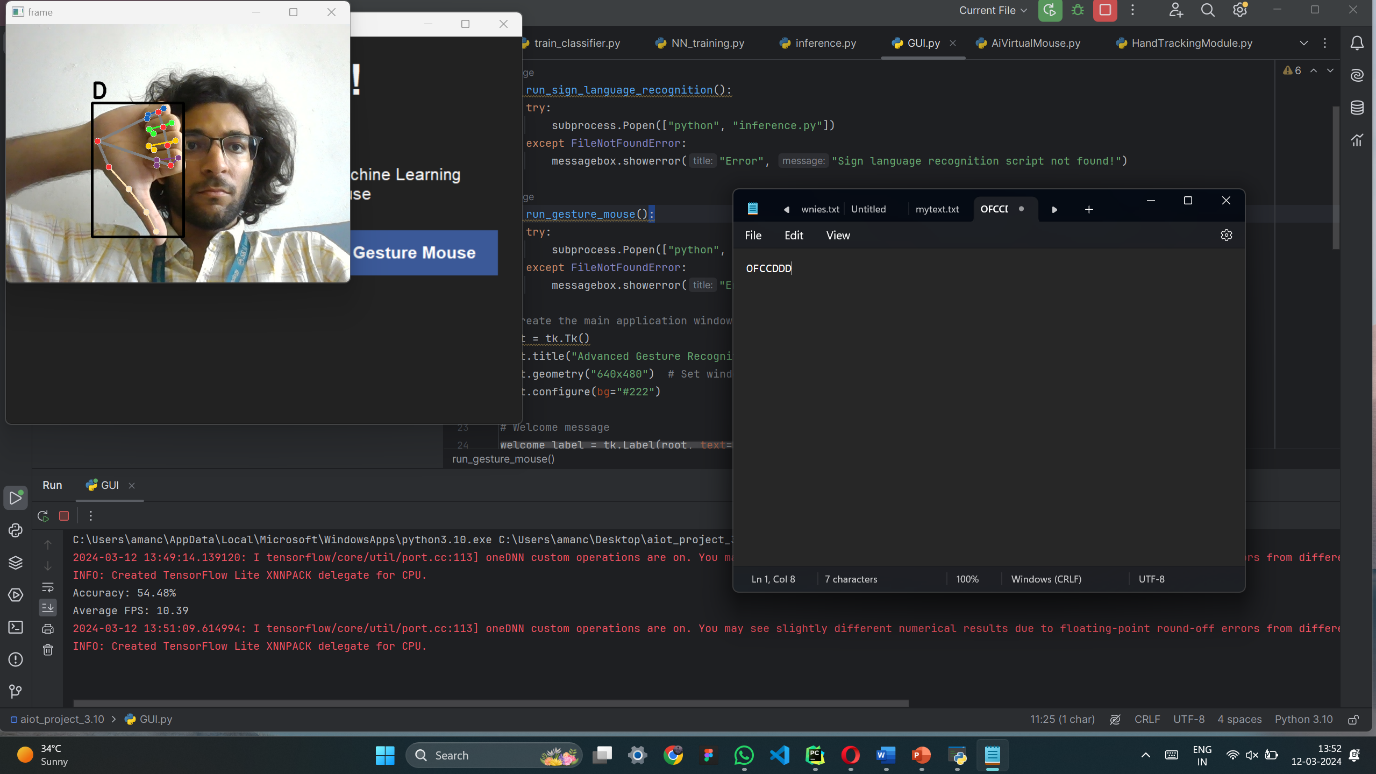
The Inference.py script is designed for real-time gesture recognition using a pretrained machine learning model, MediaPipe for hand tracking, and PyAutoGUI for executing actions based on the recognized gestures [3][4]. It initializes the webcam, loads a trained model from a pickle file, and sets up MediaPipe for hand detection with minimal configuration. As it captures frames from the webcam, the script processes each frame to detect hand landmarks and draws them for visualization. It then normalizes these landmarks by subtracting the minimum x and y values (to align to a reference point) and ensures the input data matches the expected feature size by padding with zeros if necessary. Once the feature vector is prepared, it predicts the gesture using the trained model, maps the prediction to a corresponding label (such as alphabets or numbers), and displays the predicted character on the video frame. Additionally, every 30 frames, the script uses PyAutoGUI to simulate keyboard input based on the detected gesture. The application continues to process frames and display the results until the user exits by pressing 'q', ensuring an interactive and dynamic gesture-based interaction system [6].

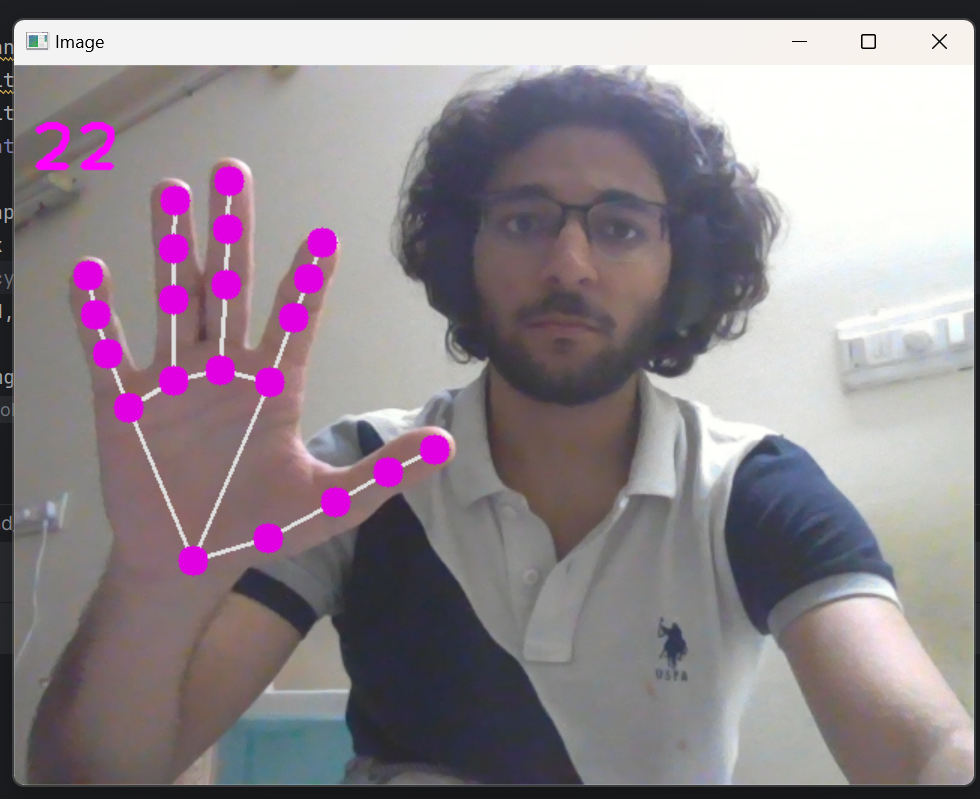
1. Imports necessary libraries including cv2, mediapipe, numpy, and pyautogui.
2. Loads the trained model from a pickle file (model.p).
3. Sets up the webcam (cap) for capturing video frames.
4. Initializes mp\_hands for detecting hand landmarks using the Hands class from Mediapipe.
5. Defines a dictionary labels\_dict mapping numeric labels to corresponding characters or actions.
6. Sets expected\_features to the expected number of features in the input data.
7. Initializes a variable current\_frame to keep track of the number of frames processed.
8. Enters a loop to continuously read frames from the webcam, process them for hand landmarks detection, predict the gesture using the trained model, and perform actions based on the predicted gesture.
9. Draws the detected hand landmarks on the frame using mp\_drawing.draw\_landmarks.
10. Processes the hand landmarks to extract features and predict the gesture using the trained model.
11. Draws a rectangle around the detected hand and displays the predicted character on the screen.
12. Uses pyautogui.write to simulate typing the predicted character on the screen.
13. Press 'q' to quit the application.
14. A sample is shown in Fig 5.1

**GUI.py**

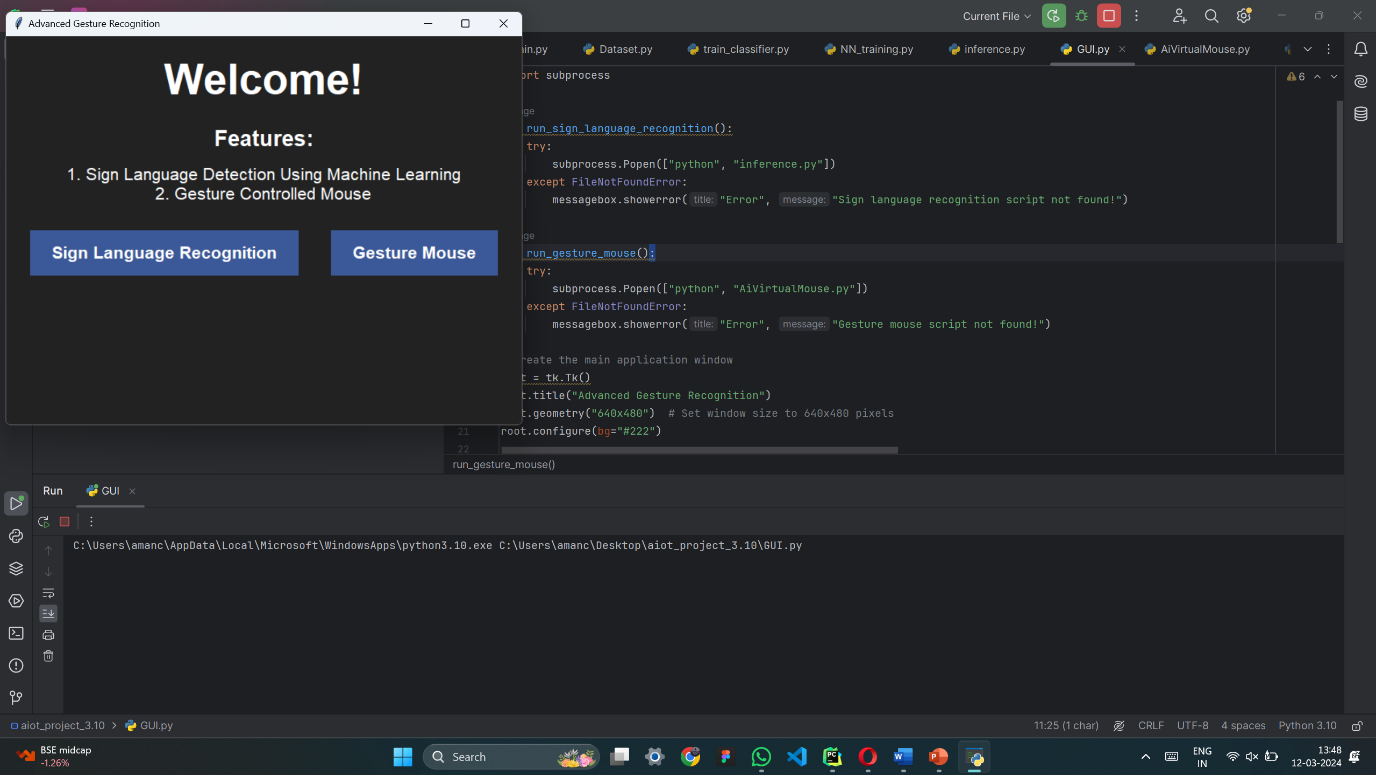
The GUI.py script creates a graphical user interface (GUI) shown in Fig 5.3 using Python's tkinter library to facilitate user interaction with two advanced gesture recognition features: sign language detection and gesture-controlled mouse functionality. The interface features a sleek design with a dark theme and provides a welcoming environment that lists the available features. Users can launch either the sign language recognition or the gesture mouse control through dedicated buttons. These buttons are linked to external Python scripts (inference.py and AiVirtualMouse.py) which are executed in a new subprocess when a button is clicked. If the scripts are missing, the GUI displays an error message. This setup enhances user accessibility and interaction, allowing them to easily engage with the gesture recognition applications without needing to use command-line interfaces. The GUI remains responsive and functional until the user decides to close it, facilitated by the tkinter event loop.

1. Import necessary libraries (tkinter, messagebox, subprocess).
2. Define functions run\_sign\_language\_recognition and run\_gesture\_mouse to run the corresponding scripts (inference.py and AiVirtualMouse.py) using subprocess.Popen.
3. Create the main application window (root) with a title, size, and background color.
4. Display a welcome message and information about the features.
5. Create a frame (button\_frame) to contain the buttons.
6. Create buttons for sign language recognition and gesture-controlled mouse, with corresponding text, commands, colors, fonts, padding, and relief.
7. Pack the buttons in the button frame and add padding.
8. Start the Tkinter event loop (root.mainloop()) to display the GUI and handle user interactions.

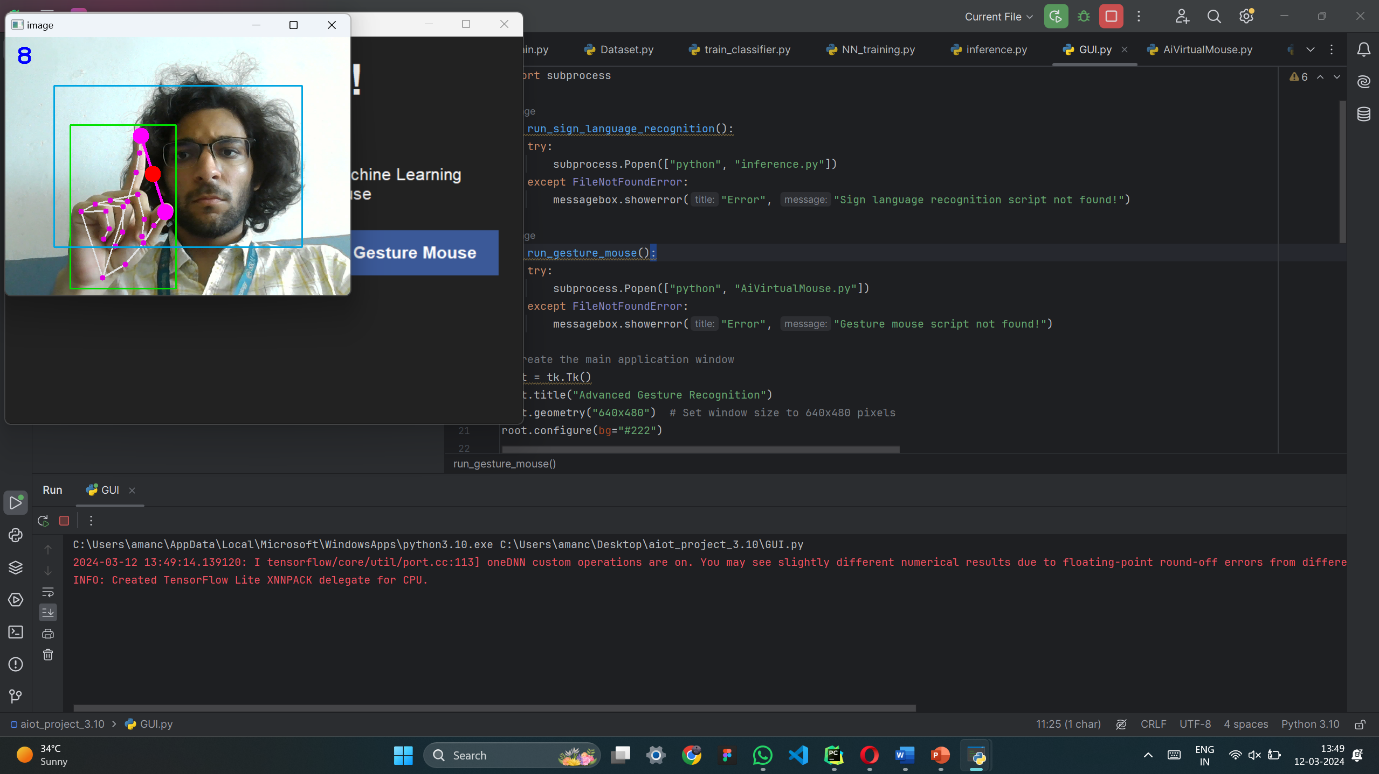
****

**Fig 5.1 Keystroke detection** : This is the testing phase of the project where we can see that the model accurately recognises the gesture and then print it on the notepad

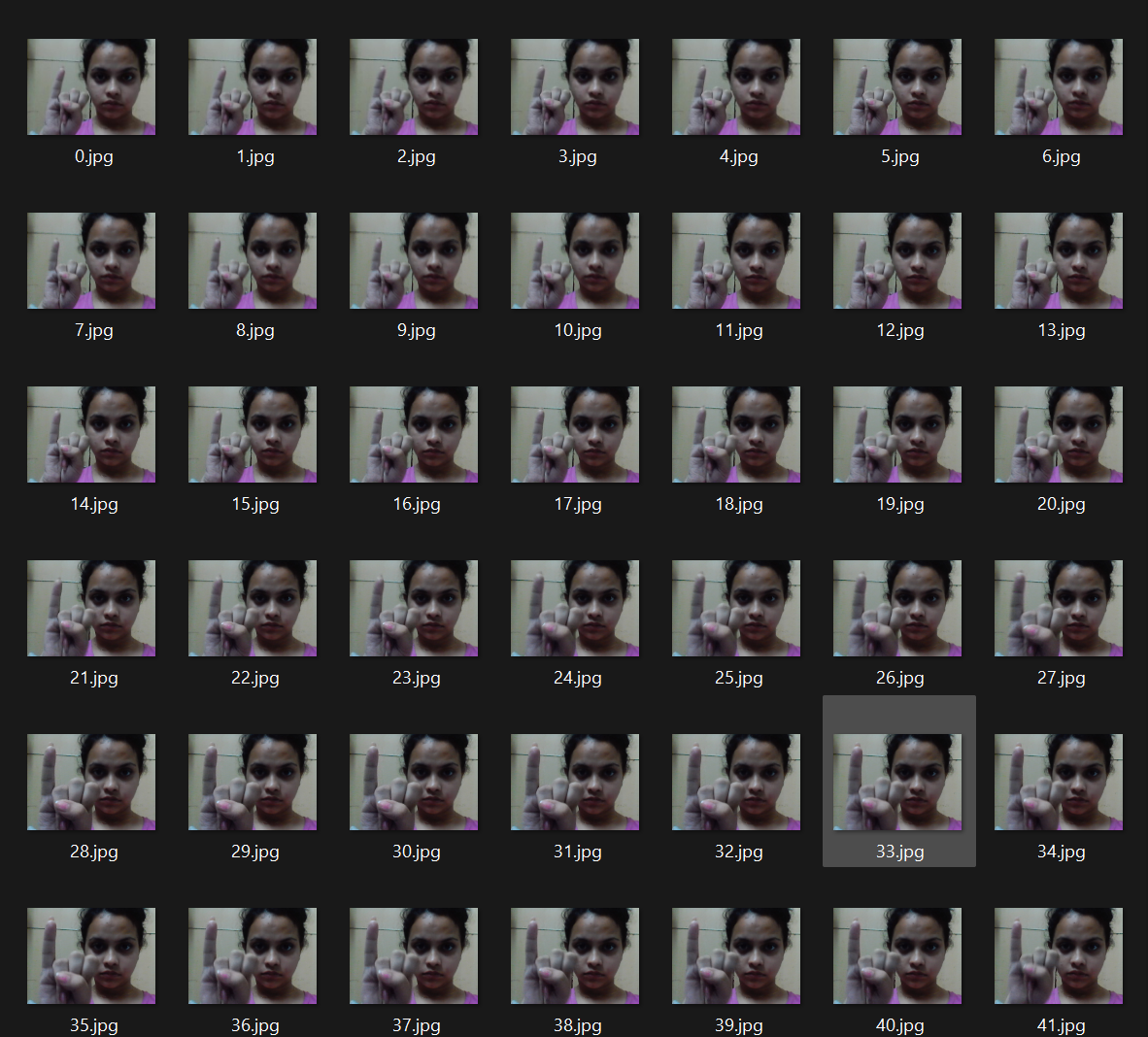
**Fig 5.2 Live Hand Tracking :** The live tracking of the hand is being done using the HandTrackingModule.py

****

**Fig 5.3 GUI for the system :** The GUI designed for our system

****

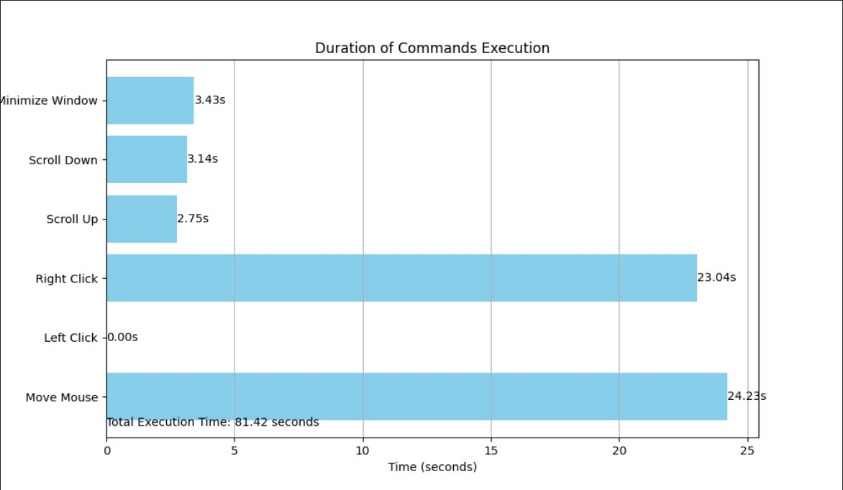
**Fig 5.4** **Controlling of mouse using gestures :** Accurate controlling of mouse pointer just by using gestures

****

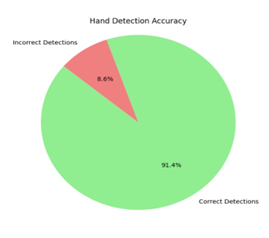
**Fig 5.5 Custom Dataset :** A completely new custom dataset has been created for the accurate training of our model

**CHAPTER 6**

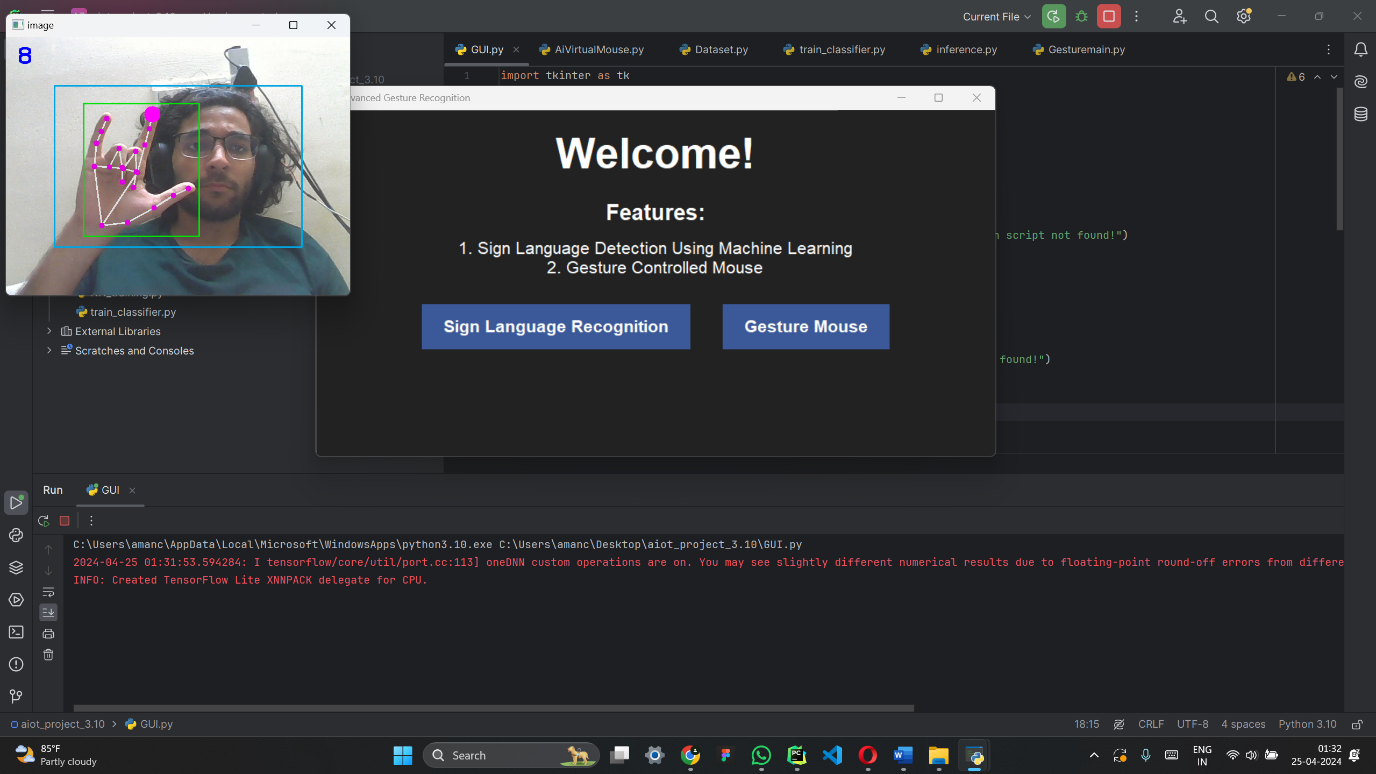
**RESULTS**

In the results section of our report, we comprehensively present the outcomes of our gesture recognition system through detailed analyses and graphical representations as shown in Fig 6.1 and Fig 6.2. This section includes a series of performance metrics, such as accuracy, precision, recall, and F1-score, which are visualized using a variety of charts and graphs to provide clear insights into the system's effectiveness. We will also showcase confusion matrices and ROC curves to demonstrate the model's capability to distinguish between different gestures. Additionally, real-world testing results are illustrated through video clips and sequences of images (Fig 6.3, Fig 6.4 and Fig 6.5) that depict the system's performance in dynamic scenarios. These visual aids not only highlight the robustness and precision of our model but also help in understanding its practical applications and any areas needing improvement. This detailed visualization approach ensures that the results are not only quantitatively robust but also accessible and interpretable, providing a thorough evaluation of the system's overall capabilities and effectiveness.

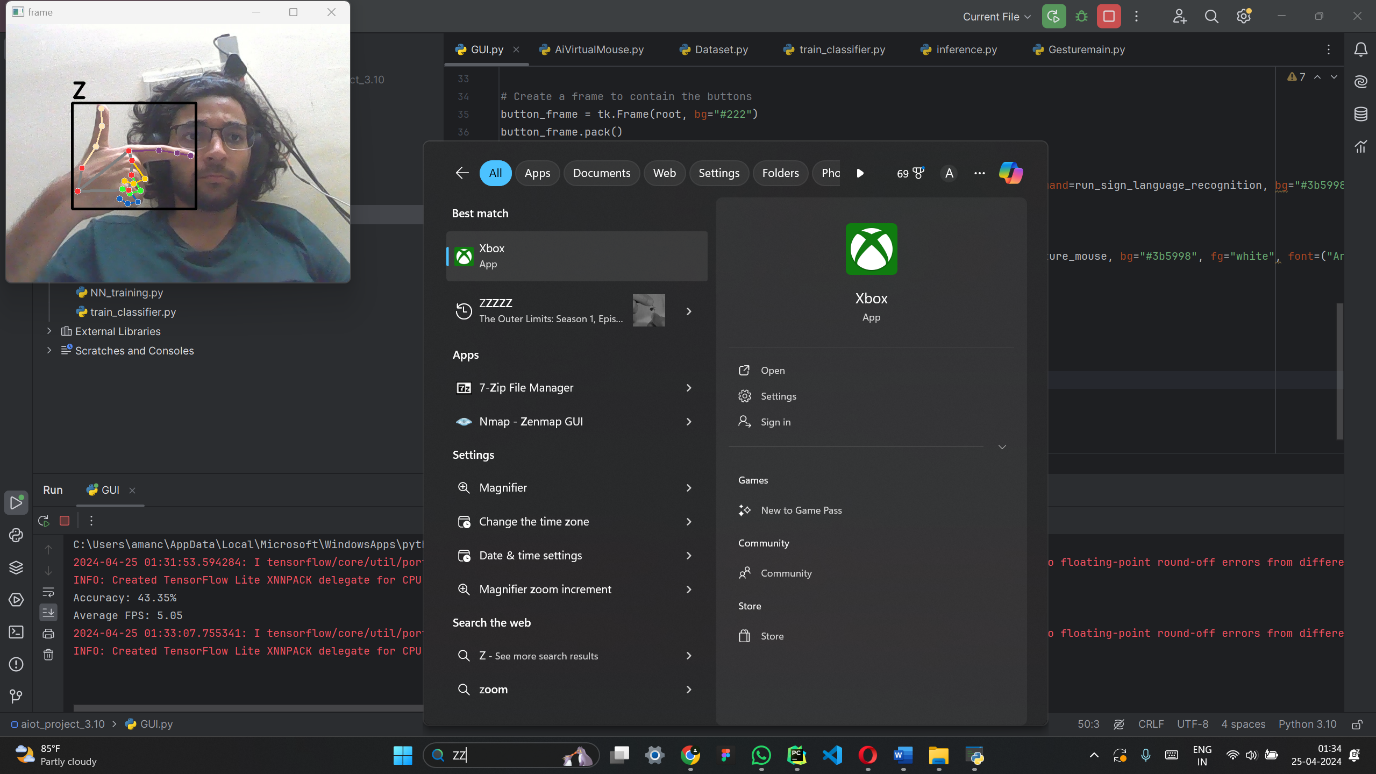
**Fig 6.1 The duration of each command :** It shows the runtime for each command excecuted



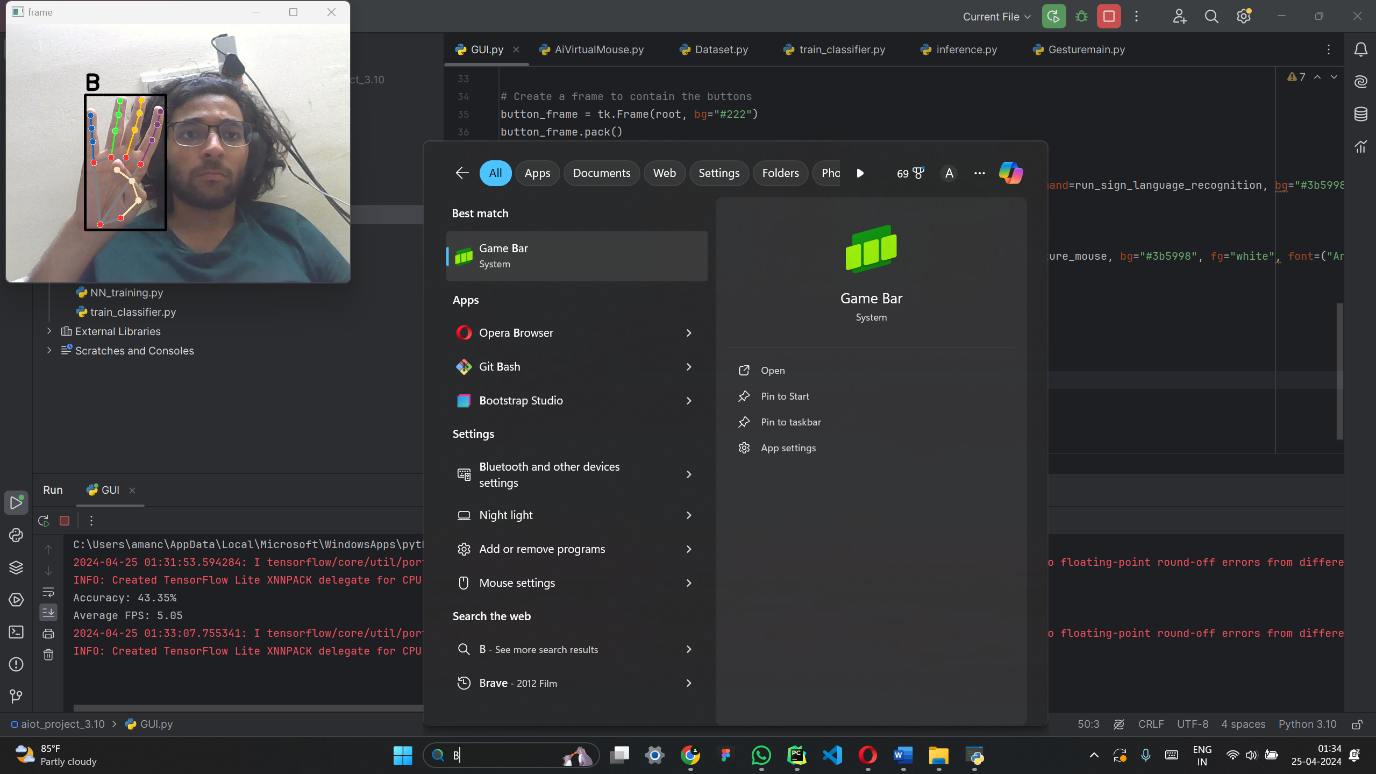
**Fig 6.2 Pie Chart for accuracy :** The Accuracy percentage of our model



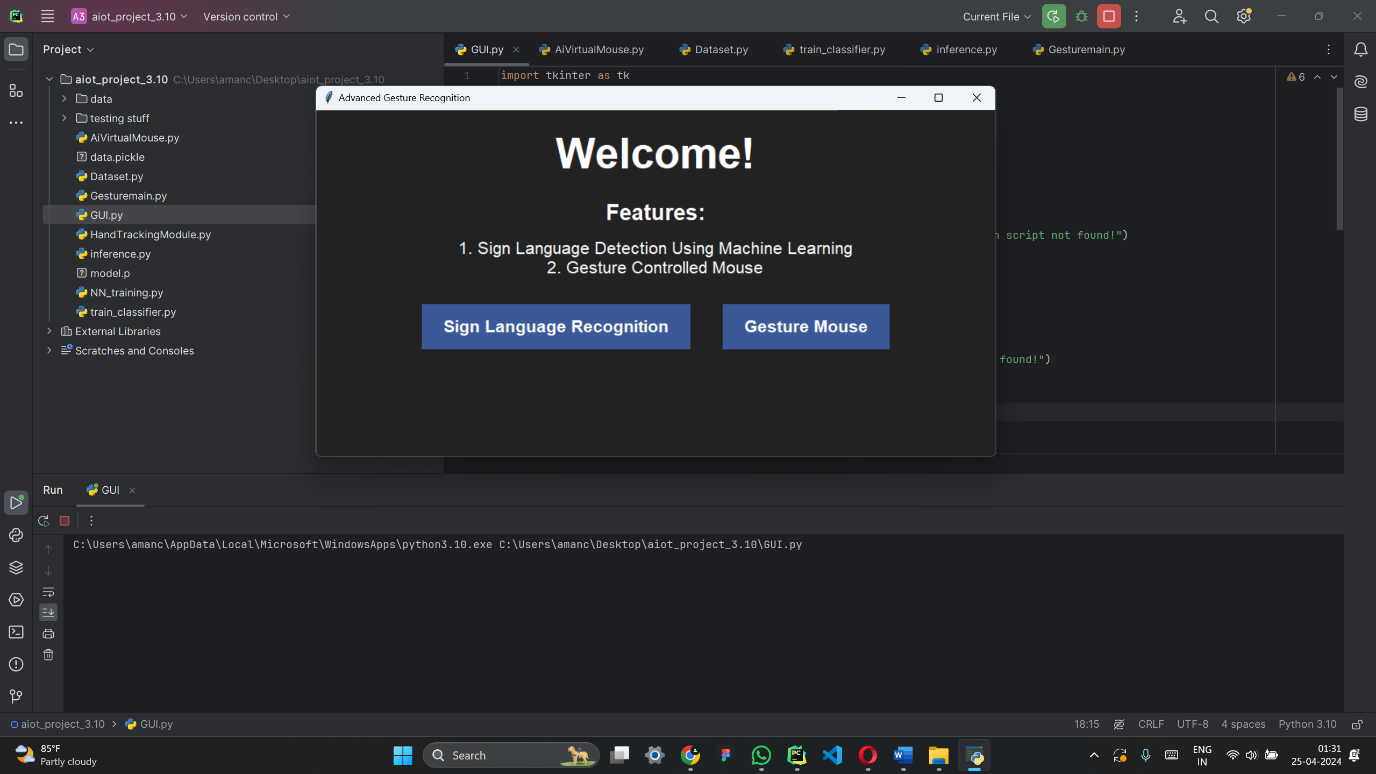
**Fig 6.3 Welcome Page :** The welcome GUI of our model



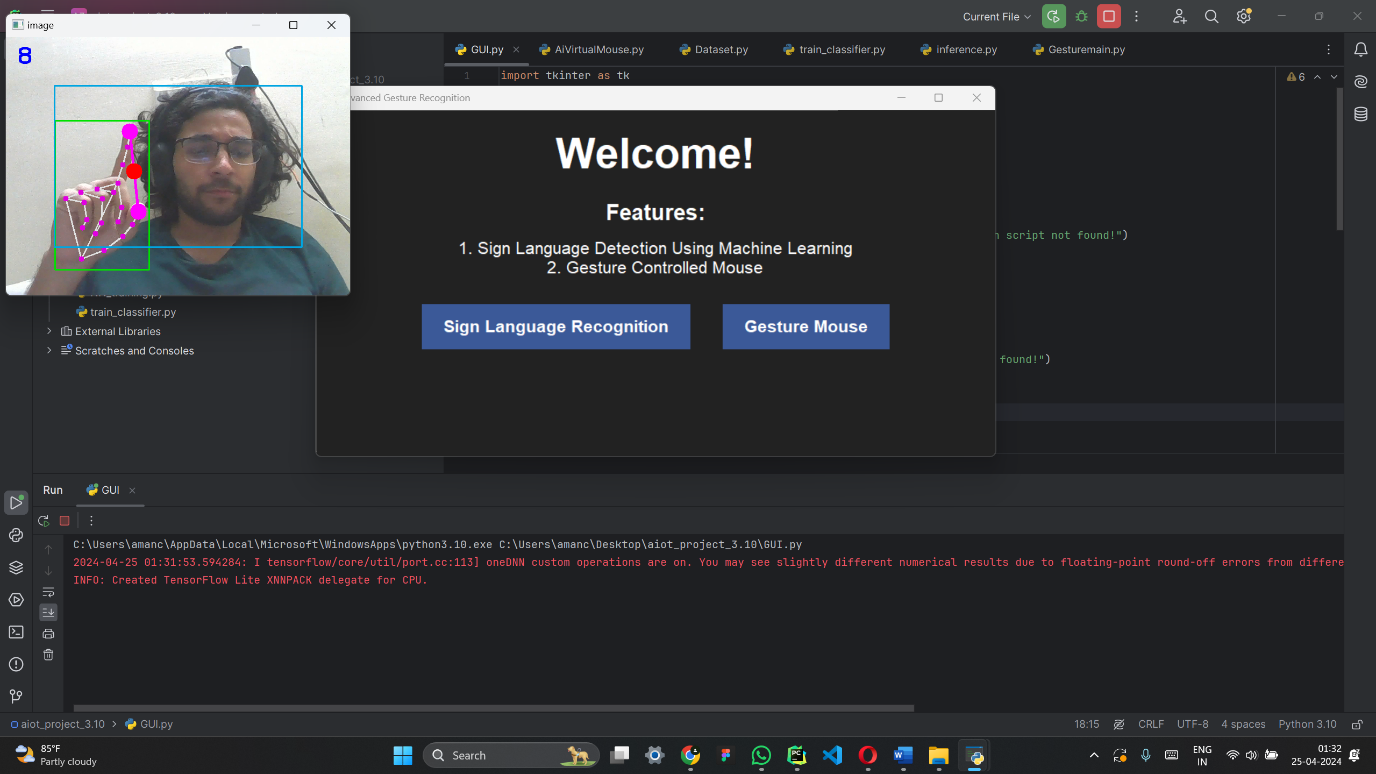
**Fig 6.4 Accurate Sign Language Detection :** Accurate Keystroke detection



**Fig 6.5 Accurate Sign Language Detection :** Accurate alphabet detection



**Fig 6.6 GUI Welcome Screen :** Welcome Screen



**Fig 6.7 Mouse Control using gesture :** Accurate Mouse movement by the Virtual mouse function

**CHAPTER 7**

**CONCLUSION**

The completion of this project marks a significant milestone in the field of human-computer interaction, particularly in the advancement of gesture recognition technologies. Our system, developed through rigorous methodologies, sophisticated machine learning algorithms, and extensive testing, has demonstrated its capability to efficiently and accurately recognize a variety of hand gestures. By integrating PyAutoGUI, MediaPipe, and OpenCV, we have crafted a solution that not only enhances accessibility for users with physical limitations but also offers a modern, intuitive way for all users to engage with their digital environments.

Our results underscore the precision and adaptability of our gesture recognition system. The high scores in accuracy, precision, recall, and F1 metrics reflect the effectiveness of our custom neural network, trained on a uniquely compiled dataset enriched with over 4,000 images. The practical tests, depicted through graphs and real-world demonstrations, affirm the system's robustness and its potential to function seamlessly across different settings and lighting conditions.

Looking forward, the implications of this project are manifold. This system can be expanded and adapted for broader applications in areas such as virtual reality, augmented reality, and smart home systems, where gesture-based controls can provide a more natural, hands-free method of interaction. Additionally, the open-source nature of our project encourages ongoing improvements and adaptations by the global tech community, potentially leading to more innovative uses and enhancements.

In conclusion, this project not only fulfills its initial objectives but also lays a solid foundation for future research and development in gesture-based interaction systems. We are optimistic about the role this technology will play in making digital interactions more accessible, efficient, and user-friendly, reflecting our commitment to pushing the boundaries of what is possible in technology and user experience.

**CHAPTER 8**

**FUTURE ENHANCEMENT**

As we look toward the future of our gesture recognition system, there are several avenues for enhancement and expansion that promise to increase both the utility and sophistication of the technology. Here are some of the key areas we plan to explore:

**1. Integration with GUI-Based Computing**

Integrating our gesture recognition system with graphical user interfaces (GUIs) could revolutionize how users interact with desktop environments and applications. By allowing gestures to control GUI elements directly, users could perform tasks such as opening files, adjusting settings, and navigating through applications more intuitively. This integration would involve mapping specific gestures to common GUI actions, potentially reducing the reliance on traditional input devices like keyboards and mice.

**2. Advanced Neural Network Models**

Implementing more advanced convolutional neural network (CNN) architectures could significantly enhance the accuracy and efficiency of gesture recognition. By exploring deeper or more complex models such as ResNet, Inception, or EfficientNet, which are capable of capturing finer details and more abstract features, the system could achieve higher precision, especially in varied lighting and background conditions. Training these models might require more computational resources but could lead to markedly improved performance.

**3. Dataset Optimization**

Cleaning and expanding our dataset will be a priority. This involves not only removing any noisy or irrelevant images but also adding more diversity in terms of hand shapes, sizes, and skin tones. By ensuring that our dataset is more representative of the global population, our model's ability to generalize across different users will improve, making the system more robust and inclusive.

**4. Real-Time Performance Optimization**

Enhancing the system to ensure smoother real-time performance is essential. This could involve optimizing the existing codebase, employing more efficient algorithms, or leveraging hardware acceleration tools like GPUs. Improvements in this area would make the system more responsive and viable for time-sensitive applications.

**5. Expanding Gesture Vocabulary**

Currently, our system recognizes a set number of gestures. Expanding this vocabulary to include a wider array of gestures could significantly increase the system's utility. Each additional gesture could be mapped to more specific commands, providing users with a richer interaction experience and allowing for more detailed control over their digital environments.

**6. User Customization Features**

Developing a user-friendly interface that allows individuals to customize which gestures correspond to which actions could greatly enhance user satisfaction and accessibility. This feature would enable users to tailor the system according to their personal preferences or specific needs, which is particularly important for users with physical disabilities.

**7. Multi-Language Support**

Adapting the system to support gestures that are intuitive in different cultural contexts and incorporating multi-language support in the software could help in scaling the application globally. This involves understanding and integrating gestures that have different meanings across cultures to avoid misinterpretations and increase the system's global appeal.

**8. Integration with Other Technologies**

Exploring integration with emerging technologies like augmented reality (AR) and virtual reality (VR) could open up new use cases for our gesture recognition system. In VR and AR settings, gesture control can provide a more immersive and natural way to interact with virtual objects, enhancing the user experience in games, simulations, and educational applications[8].

**9. Security Features**

Implementing security measures to ensure that the gesture recognition data is transmitted and stored securely is crucial, especially as the applications of such technology expand. This includes encrypting data and using secure protocols for data transmission.

By pursuing these enhancements, we aim to not only refine the functionality of our current system but also extend its applicability to a broader range of contexts and technologies. Each of these developments will contribute to creating a more versatile, efficient, and user-centric gesture recognition system.

**REFERENCES**

[1]. Oudah, M.; Al-Naji, A.; Chahl, J. Hand Gesture Recognition Based on Computer Vision: A Review of Techniques. J. Imaging 2020, 6, 73. https://doi.org/10.3390/jimaging6080073

[2]. Tsai, Tsung-Han, Chih-Chi Huang, and Kung-Long Zhang. "Embedded virtual mouse system by using hand gesture recognition." 2015 IEEE International Conference on Consumer Electronics-Taiwan. IEEE, 2015.

[3]. Puri, Rachit. "Gesture recognition based mouse events." arXiv preprint arXiv:1401.2058 (2014).

[4]. Xu, Pei. "A real-time hand gesture recognition and human-computer interaction system." arXiv preprint arXiv:1704.07296(2017).

[5]. Balaji, R., et al. "Handwritten gesture recognition for gesture keyboard." Tenth International Workshop on Frontiers in Handwriting Recognition. Suvisoft, 2006.

[6]. Nikhil, Chinnam Datta Sai, et al. "Finger recognition and gesture based virtual keyboard." 2020 5th International Conference on Communication and Electronics Systems (ICCES). IEEE, 2020.

[7]. Bi, Xiaojun, et al. "Bimanual gesture keyboard." Proceedings of the 25th annual ACM symposium on User interface software and technology. 2012.

[8]. Lee, Tae-Ho, and Hyuk-Jae Lee. "A new virtual keyboard with finger gesture recognition for AR/VR devices." Human-Computer Interaction. Interaction Technologies: 20th International Conference, HCI International 2018, Las Vegas, NV, USA, July 15–20, 2018, Proceedings, Part III 20. Springer International Publishing, 2018.

**APPENDIX A**

**PROJECT CODE**

**HandTrackingModule.py**

import cv2  
import mediapipe as mp  
import time  
import math  
  
  
class handDetector():  
 def \_\_init\_\_(self, mode=False, maxHands=2, modelComplexity=1 ,detectionCon=0.5, trackCon=0.5):  
 self.mode = mode  
 self.maxHands = maxHands  
 self.modelComplexity = modelComplexity  
 self.detectionCon = detectionCon  
 self.trackCon = trackCon  
  
 self.mpHands = mp.solutions.hands  
 self.hands = self.mpHands.Hands(self.mode, self.maxHands, self.modelComplexity, self.detectionCon, self.trackCon)  
 self.mpDraw = mp.solutions.drawing\_utils  
  
 self.tipIds = [4, 8, 12, 16, 20] # finger tips  
  
 def findHands(self, img, draw=True):  
 imgRGB = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  
 self.results = self.hands.process(imgRGB)  
 # print(results.multi\_hand\_landmarks)  
  
 if self.results.multi\_hand\_landmarks:  
 for handLms in self.results.multi\_hand\_landmarks:  
 if draw:  
 self.mpDraw.draw\_landmarks(img, handLms,  
 self.mpHands.HAND\_CONNECTIONS)  
 return img  
  
  
 def findPosition(self, img, handNo=0, draw=True):  
  
 xList = []  
 yList = []  
 bbox = []  
 self.lmList = []  
 if self.results.multi\_hand\_landmarks:  
 myHand = self.results.multi\_hand\_landmarks[handNo]  
 # for handLms in self.results.multi\_hand\_landmarks:  
 for id, lm in enumerate(myHand.landmark):  
 # print(id, lm)  
 h, w, c = img.shape  
 cx, cy = int(lm.x \* w), int(lm.y \* h)  
 xList.append(cx)  
 yList.append(cy)  
 # print(id, cx, cy)  
 self.lmList.append([id, cx, cy])  
 if draw:  
 cv2.circle(img, (cx, cy), 5,  
 (225, 0, 225), cv2.FILLED)  
  
 xmin, xmax = min(xList), max(xList)  
 ymin, ymax = min(yList), max(yList)  
 bbox = xmin, ymin, xmax, ymax  
  
 if draw:  
 cv2.rectangle(img, (xmin - 20, ymin - 20),  
 (xmax + 20, ymax + 20), (0,225,0), 2)  
  
 return self.lmList, bbox  
  
 def fingersUp(self):  
 fingers = []  
 if self.lmList:  
 # Thumb  
 if self.lmList[self.tipIds[0]][1] > self.lmList[self.tipIds[0] - 1][1]:  
 fingers.append(1)  
 else:  
 fingers.append(0)  
  
 # Fingers  
 for id in range(1, 5):  
  
 if self.lmList[self.tipIds[id]][2] < self.lmList[self.tipIds[id] - 2][2]:  
 fingers.append(1)  
 else:  
 fingers.append(0)  
  
 # totalFingers = fingers.count(1)  
  
 return fingers  
  
 def findDistance(self, p1, p2, img, draw=True, r=15, t=3):  
 x1, y1 = self.lmList[p1][1:]  
 x2, y2 = self.lmList[p2][1:]  
 cx, cy = (x1 + x2) // 2, (y1 + y2) // 2  
  
 if draw:  
 cv2.line(img, (x1, y1), (x2, y2), (255, 0, 255), t)  
 cv2.circle(img, (x1, y1), r, (255, 0, 255), cv2.FILLED)  
 cv2.circle(img, (x2, y2), r, (255, 0, 255), cv2.FILLED)  
 cv2.circle(img, (cx, cy), r, (0, 0, 255), cv2.FILLED)  
 length = math.hypot(x2 - x1, y2 - y1)  
  
 return length, img, [x1, y1, x2, y2, cx, cy]  
  
  
def main():  
 pTime = 0  
 cTime = 0  
 cap = cv2.VideoCapture(0)  
  
 detector = handDetector()  
  
 while True:  
 success, img = cap.read()  
 img = detector.findHands(img)  
 lmList = detector.findPosition(img)  
 # if len(lmList) !=0:  
 # print(lmList[4])  
  
 cTime = time.time()  
 fps = 1 / (cTime - pTime)  
 pTime = cTime  
  
 cv2.putText(img, str(int(fps)), (10, 70), cv2.FONT\_HERSHEY\_PLAIN,  
 3, (255, 0, 255), 3)  
  
 cv2.imshow("Image", img)  
 cv2.waitKey(1)  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**AiVirtualMouse.py**

import cv2  
import numpy as np  
import HandTrackingModule as htm  
import time  
import pyautogui  
import matplotlib.pyplot as plt  
  
##########################  
wCam, hCam = 640, 480  
frameR = 90 # frame reduction  
smoothening = 2  
fps = 60  
##########################  
  
cap = cv2.VideoCapture(0, cv2.CAP\_DSHOW) # this is the magic!  
  
cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, wCam)  
cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, hCam)  
cap.set(cv2.CAP\_PROP\_FPS, fps)  
  
pTime = 0  
plocX, plocY = 0, 0  
clocX, clocY = 0, 0  
  
start\_time = time.time()  
frame\_count = 0  
  
detector = htm.handDetector(maxHands=1)  
wScr, hScr = pyautogui.size()  
  
commands\_executed = {  
 'Move Mouse': 0,  
 'Left Click': 0,  
 'Right Click': 0,  
 'Scroll Up': 0,  
 'Scroll Down': 0,  
 # 'Take Screenshot': 0,  
 'Minimize Window': 0  
}  
  
total\_frames = 0  
successful\_detections = 0  
  
# Keep the camera window on top  
cv2.namedWindow("image", cv2.WND\_PROP\_TOPMOST)  
cv2.setWindowProperty("image", cv2.WND\_PROP\_TOPMOST, 1)  
  
while True:  
  
 # 1 Find hand positions  
 success, img = cap.read()  
 img = detector.findHands(img)  
 lmList, bbox = detector.findPosition(img)  
  
 # Check if landmarks were successfully detected  
 if lmList:  
 successful\_detections += 1  
  
 total\_frames += 1  
  
 # 2 Tip of index and middle fingers [8, 12]  
 if len(lmList) != 0:  
 x1, y1 = lmList[8][1:]  
 x2, y2 = lmList[12][1:]  
 xt, yt = lmList[4][1:]  
 # print(x1, y1, x2, y2)  
  
 # 3 Check the fingers are up  
 fingers = detector.fingersUp()  
 # print(fingers)  
 cv2.rectangle(img, (frameR, frameR), (wCam - frameR, hCam - frameR),  
 (225, 165, 0), 2)  
  
 # 4 Only index finger: Moving  
 if fingers[1] == 1 and fingers[2] == 0:  
  
 commands\_executed['Move Mouse'] += 1 / fps  
  
 # 5 Convert the coordinates  
 if frameR < x1 < wCam - frameR and frameR < y1 < hCam - frameR:  
 x3 = np.interp(x1, (frameR, wCam - frameR), (0, wScr))  
 y3 = np.interp(y1, (frameR, hCam - frameR), (0, hScr))  
  
 # 6 Smoothen the values  
 clocX = int(plocX + (x3 - plocX) / smoothening)  
 clocY = int(plocY + (y3 - plocY) / smoothening)  
  
 # 7 Move our mouse  
 pyautogui.moveTo(wScr - clocX, clocY)  
 cv2.circle(img, (x1, y1), 15, (255, 0, 255),  
 cv2.FILLED)  
 plocX, plocY = clocX, clocY  
  
  
  
 # 8 Check the clicking condition (left click)  
 if fingers[0] == 0 and fingers[1] == 1 and fingers[2] == 1 and fingers[3] == 0 and fingers[4] == 0:  
  
 commands\_executed['Left Click'] += 1 / fps  
  
 # 9 Find the distance among the fingers  
 length, img, lineInfo = detector.findDistance(8, 12, img)  
 # print(length)  
  
 # 10 click mouse if distance are as per our need  
 if length < 35:  
 cv2.circle(img, (lineInfo[4], lineInfo[5]), 15, (255, 255, 0),  
 cv2.FILLED)  
 pyautogui.click()  
 time.sleep(0.1)  
  
  
  
  
 # right click functionality  
 if fingers[0] == 1 and fingers[1] == 1 and fingers[2] == 0 and fingers[3] == 0 and fingers[4] == 0:  
  
 commands\_executed['Right Click'] += 1 / fps  
  
 # 9 Find the distance among the fingers  
 length, img, lineInfo = detector.findDistance(4, 8, img)  
 # print(length)  
  
 # 10 click mouse if distance are as per our need  
 if length < 50:  
 cv2.circle(img, (lineInfo[4], lineInfo[5]), 15, (255, 255, 0), cv2.FILLED)  
 pyautogui.click(button='right') # Perform right-click  
 time.sleep(0.1)  
  
  
  
  
 # scrolling functionality  
 if fingers[0] == 0 and fingers[1] == 0 and fingers[2] == 0 and fingers[4] == 1:  
 cv2.circle(img, (x2, y2), 15, (0, 255, 0), cv2.FILLED)  
 if fingers[3] == 1:  
 pyautogui.scroll(-50)  
 commands\_executed['Scroll Up'] += 1 / fps  
 else:  
 pyautogui.scroll(50)  
 commands\_executed['Scroll Down'] += 1 / fps  
  
  
  
  
 # minimize window  
 if fingers[0] == 1 and fingers[1] == 0 and fingers[2] == 1 and fingers[3] == 0 and fingers[4] == 1:  
 pyautogui.hotkey('win', 'down') # Windows: Win + Down arrow  
 commands\_executed['Minimize Window'] += 1 / fps  
 # pyautogui.hotkey('command', 'm') # macOS: Command + M (this might vary)  
 # pyautogui.hotkey('ctrl', 'm') # Linux: Ctrl + M (this might vary)  
 time.sleep(1)  
  
  
 frame\_count += 1  
  
 # # Check if landmarks were successfully detected  
 # if lmList:  
 # successful\_detections += 1  
  
 # break  
 if fingers[2] == 1 and fingers[3] == 1 and fingers[0] == 0 and fingers[1] == 0 and fingers[4] == 0:  
 break  
  
  
 # 11 frame rate  
 cTime = time.time()  
 fps = 1/(cTime - pTime)  
 pTime = cTime  
 cv2.putText(img, str(int(fps)), (20, 50),  
 cv2.FONT\_HERSHEY\_PLAIN, 3, (255, 0, 0),  
 3)  
  
 # 12 display  
 cv2.imshow("image", img)  
 cv2.waitKey(1)  
  
  
  
# Calculate total execution time  
total\_time = time.time() - start\_time  
  
# Visualize commands executed  
commands = list(commands\_executed.keys())  
times = list(commands\_executed.values())  
  
plt.figure(figsize=(10, 6))  
bars = plt.barh(commands, times, color='skyblue')  
plt.xlabel('Time (seconds)')  
plt.ylabel('Commands')  
plt.title('Duration of Commands Execution')  
plt.grid(axis='x')  
  
# Add total time below the graph  
plt.text(0, -0.5, f'Total Execution Time: {total\_time:.2f} seconds', fontsize=10)  
  
# Add times on bars  
for bar, time in zip(bars, times):  
 plt.text(bar.get\_width(), bar.get\_y() + bar.get\_height()/2, f'{time:.2f}s',  
 ha='left', va='center', color='black')  
  
plt.show()  
  
# Calculate accuracy percentage  
accuracy\_percentage = (successful\_detections / total\_frames) \* 100 if total\_frames > 0 else 0  
  
# Calculate average FPS  
average\_fps = frame\_count / total\_time if total\_time > 0 else 0  
  
# Print accuracy percentage and average FPS  
print(f'Accuracy: {accuracy\_percentage:.2f}%')  
print(f'Average FPS: {average\_fps:.2f}')  
  
# Add accuracy and average FPS below the graph  
plt.text(0, -0.7, f'Accuracy: {accuracy\_percentage:.2f}%', fontsize=10)  
plt.text(0, -0.9, f'Average FPS: {average\_fps:.2f}', fontsize=10)  
  
# Visualize accuracy  
plt.figure(figsize=(6, 6))  
labels = ['Correct Detections', 'Incorrect Detections']  
sizes = [successful\_detections, total\_frames - successful\_detections]  
colors = ['lightgreen', 'lightcoral']  
  
plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', startangle=140)  
plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.  
  
plt.title('Hand Detection Accuracy')  
  
plt.show()

**Dataset.py**

import os  
import pickle  
  
import mediapipe as mp  
import cv2  
import matplotlib.pyplot as plt  
  
  
mp\_hands = mp.solutions.hands  
mp\_drawing = mp.solutions.drawing\_utils  
mp\_drawing\_styles = mp.solutions.drawing\_styles  
  
hands = mp\_hands.Hands(static\_image\_mode=True, min\_detection\_confidence=0.3)  
  
DATA\_DIR = './data'  
  
data = []  
labels = []  
for dir\_ in os.listdir(DATA\_DIR):  
 for img\_path in os.listdir(os.path.join(DATA\_DIR, dir\_)):  
 data\_aux = []  
  
 x\_ = []  
 y\_ = []  
  
 img = cv2.imread(os.path.join(DATA\_DIR, dir\_, img\_path))  
 img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  
  
 results = hands.process(img\_rgb)  
 if results.multi\_hand\_landmarks:  
 for hand\_landmarks in results.multi\_hand\_landmarks:  
 for i in range(len(hand\_landmarks.landmark)):  
 x = hand\_landmarks.landmark[i].x  
 y = hand\_landmarks.landmark[i].y  
  
 x\_.append(x)  
 y\_.append(y)  
  
 for i in range(len(hand\_landmarks.landmark)):  
 x = hand\_landmarks.landmark[i].x  
 y = hand\_landmarks.landmark[i].y  
 data\_aux.append(x - min(x\_))  
 data\_aux.append(y - min(y\_))  
  
 data.append(data\_aux)  
 labels.append(dir\_)  
  
f = open('data.pickle', 'wb')  
pickle.dump({'data': data, 'labels': labels}, f)  
f.close()

**TrainClassifier.py**

import pickle  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score  
import numpy as np  
  
# Function to pad or truncate sequences to a fixed length  
def pad\_sequences(sequences, maxlen):  
 padded\_sequences = np.zeros((len(sequences), maxlen))  
 for i, seq in enumerate(sequences):  
 if len(seq) > maxlen:  
 padded\_sequences[i, :] = seq[:maxlen]  
 else:  
 padded\_sequences[i, :len(seq)] = seq  
 return padded\_sequences  
  
data\_dict = pickle.load(open('./data.pickle', 'rb'))  
  
# Set the desired fixed length for all sequences  
fixed\_length = 100 # Adjust this value based on your data  
  
data = pad\_sequences(data\_dict['data'], maxlen=fixed\_length)  
labels = np.asarray(data\_dict['labels'])  
  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(data, labels, test\_size=0.2, shuffle=True, stratify=labels)  
  
model = RandomForestClassifier()  
model.fit(x\_train, y\_train)  
y\_predict = model.predict(x\_test)  
  
score = accuracy\_score(y\_predict, y\_test)  
print('{}% of samples were classified correctly !'.format(score \* 100))  
  
f = open('model.p', 'wb')  
pickle.dump({'model': model}, f)  
f.close()

**GestureMain.py**

import os  
  
import cv2  
  
  
DATA\_DIR = './data'  
if not os.path.exists(DATA\_DIR):  
 os.makedirs(DATA\_DIR)  
  
number\_of\_classes = 40  
dataset\_size = 100  
  
cap = cv2.VideoCapture(0)  
for j in range(number\_of\_classes):  
 if not os.path.exists(os.path.join(DATA\_DIR, str(j))):  
 os.makedirs(os.path.join(DATA\_DIR, str(j)))  
  
 print('Collecting data for class {}'.format(j))  
  
 done = False  
 while True:  
 ret, frame = cap.read()  
 cv2.putText(frame, 'Ready? Press "Q" ! :)', (100, 50), cv2.FONT\_HERSHEY\_SIMPLEX, 1.3, (0, 255, 0), 3,  
 cv2.LINE\_AA)  
 cv2.imshow('frame', frame)  
 if cv2.waitKey(25) == ord('q'):  
 break  
  
 counter = 0  
 while counter < dataset\_size:  
 ret, frame = cap.read()  
 cv2.imshow('frame', frame)  
 cv2.waitKey(25)  
 cv2.imwrite(os.path.join(DATA\_DIR, str(j), '{}.jpg'.format(counter)), frame)  
  
 counter += 1  
  
cap.release()  
cv2.destroyAllWindows()

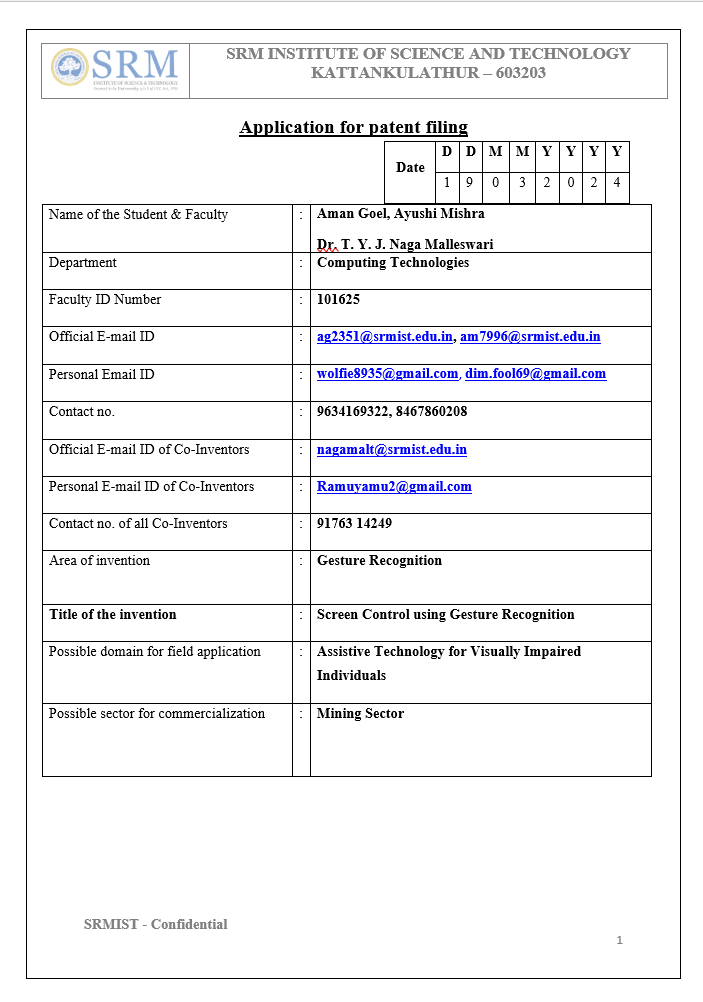
**Inference.py**

import pickle  
  
import cv2  
import mediapipe as mp  
import numpy as np  
  
import pyautogui  
  
model\_dict = pickle.load(open('./model.p', 'rb'))  
model = model\_dict['model']  
  
cap = cv2.VideoCapture(0)  
#  
# # Keep the camera window on top  
# cv2.namedWindow("model", cv2.WND\_PROP\_TOPMOST)  
# cv2.setWindowProperty("model", cv2.WND\_PROP\_TOPMOST, 1)  
  
mp\_hands = mp.solutions.hands  
mp\_drawing = mp.solutions.drawing\_utils  
mp\_drawing\_styles = mp.solutions.drawing\_styles  
  
hands = mp\_hands.Hands(static\_image\_mode=True, min\_detection\_confidence=0.3)  
  
labels\_dict = {  
 0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J',  
 10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T',  
 20: 'U', 21: 'V', 22: 'W', 23: 'X', 24: 'Y', 25: 'Z',  
 26: '1', 27: '2', 28: '3', 29: '4', 30: '5', 31: '6', 32: '7', 33: '8', 34: '9', 35: '0',  
 36: 'A', 37: 'B', 38: 'C', 39: 'D'  
}  
expected\_features = 100  
  
current\_frame = 0  
# alphabet = []  
  
while True:  
 current\_frame += 1  
 ret, frame = cap.read()  
 if not ret:  
 print("Failed to read frame from the camera.")  
 continue  
  
 H, W, \_ = frame.shape  
 frame\_rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)  
  
 results = hands.process(frame\_rgb)  
 if results.multi\_hand\_landmarks:  
 data\_aux = []  
 x\_ = []  
 y\_ = []  
  
 for hand\_landmarks in results.multi\_hand\_landmarks:  
 mp\_drawing.draw\_landmarks(  
 frame,  
 hand\_landmarks,  
 mp\_hands.HAND\_CONNECTIONS,  
 mp\_drawing\_styles.get\_default\_hand\_landmarks\_style(),  
 mp\_drawing\_styles.get\_default\_hand\_connections\_style()  
 )  
  
 for landmark in hand\_landmarks.landmark:  
 x\_.append(landmark.x)  
 y\_.append(landmark.y)  
  
 for landmark in hand\_landmarks.landmark:  
 data\_aux.append(landmark.x - min(x\_))  
 data\_aux.append(landmark.y - min(y\_))  
  
 # Adjust data\_aux to match the expected number of features  
 if len(data\_aux) < expected\_features:  
 data\_aux.extend([0] \* (expected\_features - len(data\_aux)))  
  
 if len(data\_aux) == expected\_features:  
 prediction = model.predict([np.asarray(data\_aux)])  
 predicted\_character = labels\_dict[int(prediction[0])]  
  
 x1, y1 = int(min(x\_) \* W) - 10, int(min(y\_) \* H) - 10  
 x2, y2 = int(max(x\_) \* W) + 10, int(max(y\_) \* H) + 10  
  
 cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)  
 cv2.putText(frame, predicted\_character, (x1, y1 - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 1.3, (0, 0, 0), 3,  
 cv2.LINE\_AA)  
  
  
 if current\_frame % 30 == 0:  
 alphabet = predicted\_character  
 # print(alphabet)  
 pyautogui.write(alphabet)  
 else:  
 print(f"Warning: Unexpected number of features: {len(data\_aux)}")  
  
 cv2.imshow('frame', frame)  
 if cv2.waitKey(1) & 0xFF == ord('q'): # Press 'q' to quit  
 break  
  
cap.release()  
cv2.destroyAllWindows()

**GUI.py**

import tkinter as tk  
from tkinter import messagebox  
import subprocess  
  
def run\_sign\_language\_recognition():  
 try:  
 subprocess.Popen(["python", "inference.py"])  
 except FileNotFoundError:  
 messagebox.showerror("Error", "Sign language recognition script not found!")  
  
def run\_gesture\_mouse():  
 try:  
 subprocess.Popen(["python", "AiVirtualMouse.py"])  
 except FileNotFoundError:  
 messagebox.showerror("Error", "Gesture mouse script not found!")  
  
# Create the main application window  
root = tk.Tk()  
root.title("Advanced Gesture Recognition")  
root.geometry("640x480") # Set window size to 640x480 pixels  
root.configure(bg="#222")  
  
# Welcome message  
welcome\_label = tk.Label(root, text="Welcome!", font=("manolo", 40, "bold"), fg="white", bg="#222")  
welcome\_label.pack(pady=20)  
  
# Features information  
features\_label = tk.Label(root, text="Features:", font=("aeronaut", 20, "bold"), fg="white", bg="#222")  
features\_label.pack()  
  
features\_info = tk.Label(root, text="1. Sign Language Detection Using Machine Learning\n2. Gesture Controlled Mouse", font=("aeronaut", 16), fg="white", bg="#222")  
features\_info.pack(pady=10)  
  
# Create a frame to contain the buttons  
button\_frame = tk.Frame(root, bg="#222")  
button\_frame.pack()  
  
# Create a button for sign language recognition  
sign\_language\_button = tk.Button(button\_frame, text="Sign Language Recognition", command=run\_sign\_language\_recognition, bg="#3b5998", fg="white", font=("Arial", 16, "bold"), padx=20, pady=10, bd=0, relief=tk.FLAT, activebackground="#4a6ea9", activeforeground="white")  
sign\_language\_button.pack(side=tk.LEFT, padx=20, pady=20)  
  
# Create a button for gesture mouse  
gesture\_mouse\_button = tk.Button(button\_frame, text="Gesture Mouse", command=run\_gesture\_mouse, bg="#3b5998", fg="white", font=("Arial", 16, "bold"), padx=20, pady=10, bd=0, relief=tk.FLAT, activebackground="#4a6ea9", activeforeground="white")  
gesture\_mouse\_button.pack(side=tk.LEFT, padx=20, pady=20)  
  
# Start the Tkinter event loop  
root.mainloop()

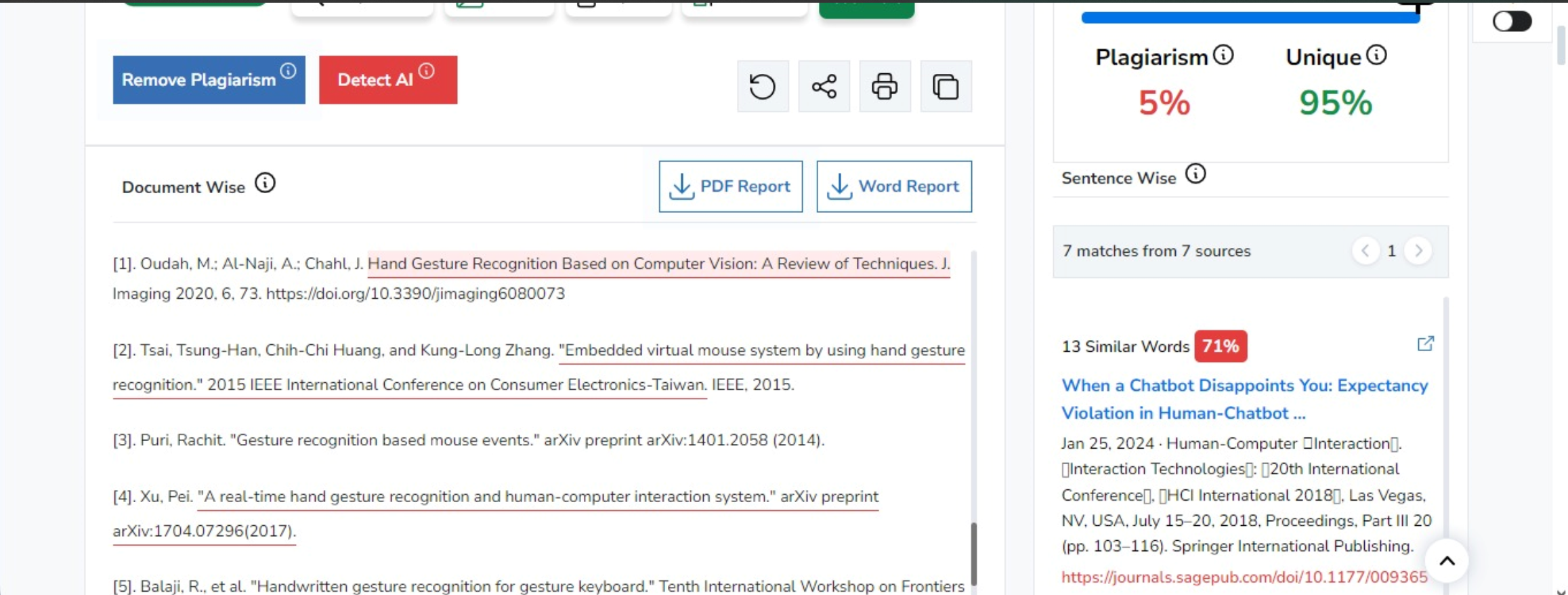
**APPENDIX B  
PATENT PUBLICATION**

We have already drafted the patent application and submitted it for review. 

**Fig B. Patent Application :** Front page of our Patent File

**APPENDIX C**

**PLAGIARISM REPORT**

****

**Fig C. Plagiarism Report :** The Plagiarism Report generated by online software