*Dynamic Hand Gesture Recognition using Neural Network*

Ansh Pandey   
 *Apex Institute of Technology - CSE*  
*Chandigarh University*Chandigarh, India  
20BCS6741@cuchd.in

***Abstract*— Dynamic Hand Gesture Recognition is the most popular and widely used technology in the field of Computer Vision. It let the computer understand the hand gestures performed by the user according to the data stored in its database. In this paper, a recognition system is developed which can recognize dynamic hand gestures. It has two parts: (I) Virtual Keyboard (II) Virtual Mouse. In this recognition system, we are basically recognizing the actions performed by the hand. By using CNN and RNN for recognizing the continuous motion of hand.**

***Keywords: hand gesture recognition; deep learning; convolutional neural networks, recurrent neural networks, computer vision, neural networks, human-computer interaction***

# Introduction

Hand Gesture Recognition is one of the most interesting fields of human-computer interface. Hand Gesture Recognition System recognizes the actions performed by the user and then send some information or perform some action as described or stated in its database. Human actions can be sensed using either camera or sensors. There can be two types of hand gestures, one is Static and second one is Dynamic. In static hand gesture, there we need to put the hand stable and it’s all about the shape of the hand. While in Dynamic hand gestures, there is a continuous movement of hand.

Dynamic hand gesture recognition is a subfield of hand gesture recognition that involves recognizing gestures that involve movement, as opposed to static hand poses.

Dynamic hand gesture recognition using neural networks has garnered considerable attention in recent years due to its ability to accurately recognize hand gestures in real-time, robustness to noise and occlusion, and its ability to handle variations in hand shape and size. Neural networks are machine learning algorithms inspired by the functioning of the human brain, consisting of interconnected nodes that process input data and produce output predictions.

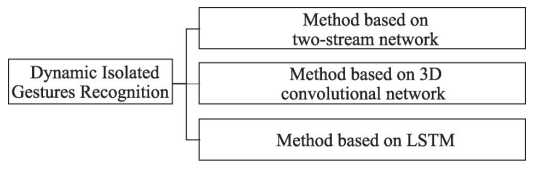
The dynamic hand gesture recognition using neural networks process involves two main stages: hand tracking and gesture recognition. In the hand tracking stage, the location of the hand is detected and tracked in real-time using computer vision techniques such as background subtraction, skin color segmentation, and feature extraction. The hand region is then segmented and normalized to a standard size and orientation.

In the gesture recognition stage, the preprocessed hand images are inputted to a neural network model that has been trained to recognize various hand gestures. The neural network learns to map the input hand images to the corresponding gesture labels by adjusting the weights and biases of the network during the training phase. The training data usually consists of a large dataset of labeled hand gesture images that are used to train the neural network model.

Various types of neural network models can be used for dynamic hand gesture recognition, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and combination models. CNNs are commonly used for image recognition tasks and are ideal for detecting spatial features in the input hand images. RNNs, on the other hand, are useful for recognizing temporal patterns in the hand gestures, making them suitable for recognizing sequential hand gestures.

The performance of the neural network model for dynamic hand gesture recognition can be evaluated using various metrics, such as accuracy, precision, recall, and F1-score. The accuracy of the model can be improved by increasing the size and diversity of the training dataset, optimizing the hyperparameters of the neural network model, and incorporating data augmentation techniques such as rotation, scaling, and translation.

In conclusion, dynamic hand gesture recognition using neural networks is an exciting technology that has the potential to revolutionize human-computer interaction and communication. It has numerous practical applications in various fields, including gaming, robotics, virtual reality, and sign language recognition. As the technology continues to evolve, we can expect to see more accurate and robust hand gesture recognition systems that can adapt to various environmental conditions and user requirements.



*Figure Source:* [*https://pdf.sciencedirectassets.com/*](https://pdf.sciencedirectassets.com/)

# Literature survey

The researches on gesture recognition were mainly carried in and after 1964, when a scientist named Ivan E. Sutherland uses a pen with button to draw a line in the year 1964.

Talking about the technologies in Deep Learning and Neural Network, there are many approaches defined to perform gesture recognition. For identifying extracted features, we can use Support Vector Machine, Decision Trees, K – Nearest Neighbor, etc.

Following are the few existing systems-

* Wearable Armband: It is a real – time gesture recognition system, developed using electromyography sensors and algorithms such as K – Nearest Neighbor and Decision Trees were also used.
* Vision – based Devices: In such devices, recognition of hands and its poses and of face is done using feature – based template matching techniques.

In the paper “Dynamic Hand Gesture Recognition System using Neural Network” by Chitralekha Mahanta, T. Srinivas Yadav, and Hemanta Medhi [1] published in 2011, a recognition system for dynamic hand gestures is proposed. In the system, the hand is segmented by using background subtraction method. MPEG-7 ART based shape descriptors are used to extract spatial information. The approach followed in the paper is based on particle filter to extract trajectory features. After collecting suitable features, Radial Basis Function neural network is used for classification. Gesture recognition rate is in the range of 80% to 98%. While preprocessing, to remove salt and pepper noise and additive Gaussian noises present in the image, median filter and mean filter is used. After this segmentation is done, the hand region extraction method is implemented by using background subtraction algorithm. For spatial information, MPEG – 7 ART shape descriptors are used. And Particle Filters are used for Tracking of Hand. Spatial feature vectors and features obtained from gesture trajectory are given as input to the neural network classifier to classify different dynamic gestures. Radial basis function (RBFs) neural network is used for classification. A total of 20 dynamic hand gestures have been chosen for our experiments. Among them, 8 belong to ASL gestures and 12 belong to control commands.

Dynamic Hand Gesture Recognition Using Computer Vision and Neural Networks by M. I. N. P. Munasinghe published in 2018, Nuwan Munasinghe has designed and developed a system which can n recognize gestures in front of a web camera real time using motion history images (MHI) and feedforward neural networks. Firstly, background from captured frames is removed using Gaussian mixture based background/foreground segmentation algorithm in order to capture moving areas in the frame and thereafter median filtering has applied to remove random noise from the frame. Then binary thresholding with Otsu’s binarization has applied and it will identify optimal threshold value and these processed frames are merged and cumulative motion history image is generated using a developed algorithm based on the structural similarity measure. Structural similarity between the cumulated image and the initial frame also calculated and used in this algorithm. Finally feed forward neural network with stochastic gradient-based optimizer has used to classify the gestures. k. The neural network is used to determine the probabilities for each category and if the category with the highest probability surpasses probability of 0.8, it is identified as a correct gesture.

Another paper by Kenneth Lai and Svetlana N. Yanushkevich “CNN+RNN Depth and Skeleton based Dynamic Hand Gesture Recognition” published in 2020 has proposed a system that consists of two main components: a depth-based CNN+RNN and a skeleton-based RNN for automated hand gesture recognition using both depth and skeleton data. The results of the conducted experimental study are summarized as follows:

1) Usage of depth data, in addition to visual spectrum, RGB data, has potential to greatly improve the performance of hand gesture recognition algorithms that rely on temporal patterns. While usage of only depth data with CNN leads to the average recognition rate of 32.24%. However, utilizing the CNN for feature extraction, and passing those features to the RNN, the overall performance is shown to increase up to 75.79%. This results indicate that CNN alone is insufficient in extracting enough information from a singular depth image to correctly predict the desired gesture. By combining the CNN with the RNN, the new network is able to recognize patterns from the features extracted from a CNN throughout multiple frames. Another aspect influencing the performance is the chosen timestep parameter which depending on the input data may be severely truncated when given long sequences or padded with blank information for short sequences.

2) The performance of using skeleton data has shown a recognition rate of 82.18% which is higher compared to the depth-based networks. However, when examining the recognition rates for each grain gesture, the depth-based approach produces a higher recognition rate for the fine – grained gesture. Based on this observation, the fusion of both depth-based and skeleton-based networks should yield a higher overall recognition rate. Experiments show that by using a score-level fusion, a recognition rate of 85.46% is achieved, which is higher than the rates of 82.18% and 76.50% for the independent skeleton and the depth based approaches, respectively. In addition to an overall higher recognition rates, the rates for each type of grain gesture also increased. For the fine grained gestures, while the rate for separately obtained depthbased and skeleton-based approach are 73.50% and 67.20%, respectively, the combined rate is 76.00%. For the coarse grained gestures, the depth-based (78.17%) and skeleton-based (89.00%), when combined, show the rate of 90.72%.

3) The applied fusion allowed to achieve very good performance for the dataset consisting of 14 gestures. When the same approach is applied to the 28 gesture version, the performance degrades to 74.19% which is an ≈11% decrease. The main reason for this decrease is that the networks’ weights are optimized for recognizing 14 gestures. Therefore, a better performance can achieved by training each network independently for recognizing 28 gestures prior to fusion.

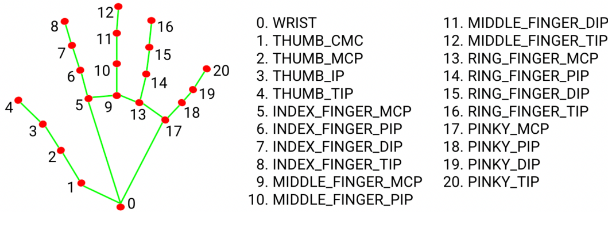
# Proposed System

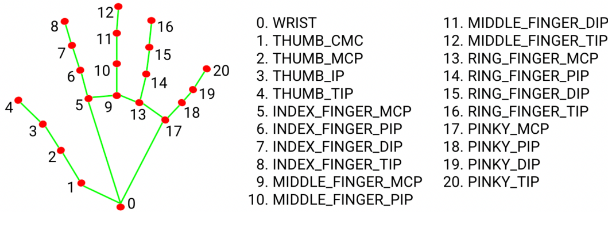
The system proposed by us will have features like Virtual Keyboard and Virtual Mouse that will help us to access the apps, searching of apps, typing, etc. on the system.

In Virtual Keyboard, we can type anything anywhere in the text area. Virtual Keyboard is necessary in a computer/laptop as sometimes we have faced issues related to keyboard that some keyboard buttons are missing or some buttons do not work, at that moment Virtual Keyboard will help us to use the system. In virtual Mouse, we can use our finger the same as a mouse but virtually. The camera will detect the movements of hand and fingers and will detect its actions and will response accordingly.

In case of Virtual Mouse, we are using hand landmark detection and hand gesture recognition like if the two fingers are connected, the mouse will click and by moving finger we can move the cursor. Hand Landmark Detection is done by Mediapipe (K – Nearest Neighbor Algorithm) and Hand Gesture Recognition is done using Convolutional Neural Network. The Mediapipe Hand Landmarker detects the landmarks of the hands. It is used to localize key points of the hands and render visual effects over the hands.

The 21 key points used by the Mediapipe Hand Landmarker are given below:





*Fig.:*[*https://developers.google.com/mediapipe/solutions/vision/hand\_landmarker*](https://developers.google.com/mediapipe/solutions/vision/hand_landmarker)

# Virtual Keyboard

the basic steps involved in creating a virtual keyboard using dynamic neural network simulation:

1. Data Collection: Collect a dataset of users typing on a virtual keyboard over an extended period of time. This dataset will be used to train the dynamic neural network model.
2. Data Preprocessing: Preprocess the data to prepare it for training. This may involve converting the keystrokes into numerical values, normalizing the data, and splitting it into training and testing sets.
3. Dynamic Neural Network Architecture: Decide on the dynamic neural network architecture to use. This may involve selecting the number of layers, the number of neurons in each layer, and the activation function to use. Additionally, the model must be designed to be able to adapt to changes in the user's typing patterns over time.
4. Training: Train the dynamic neural network model on the training dataset using an optimization algorithm such as stochastic gradient descent. The goal is to minimize the error between the predicted output and the actual output, while also allowing the model to adapt to changes in the user's typing patterns.
5. Validation: Evaluate the performance of the dynamic neural network model on a validation dataset. This helps to ensure that the model is not overfitting to the training data and is able to generalize well to new data.
6. Deployment: Deploy the dynamic neural network model to create a virtual keyboard. This involves integrating the model into a software application that allows users to type on a virtual keyboard while the model adapts to their typing patterns over time.

Overall, a virtual keyboard using dynamic neural network simulation can provide a more accurate and personalized typing experience for users, which can be particularly useful for individuals who have difficulty typing on a physical keyboard or for situations where a physical keyboard is not available.

Top of Form

Explanation:

that implements a virtual keyboard using the Mediapipe library for hand tracking and the pynput library for controlling keyboard input. The program uses a webcam to capture images, and applies hand tracking using the Mediapipe Hands module to detect the position of the user's hand.

The program defines a class called Store which is used to store the position, size, and text of each button on the virtual keyboard. The program defines a list of buttons by iterating over the keys list, and creating a Store object for each key.

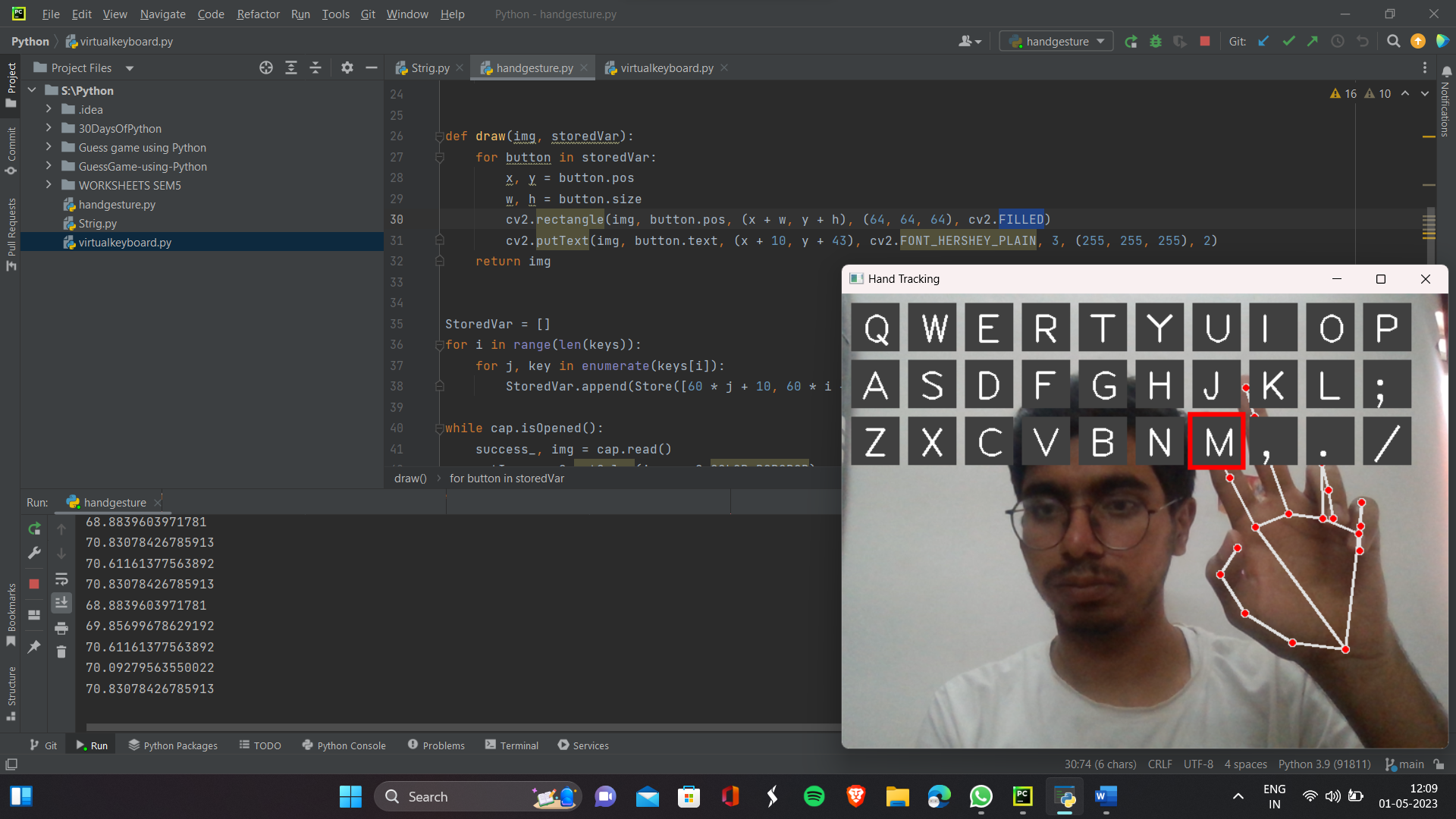
The program then enters a loop where it reads frames from the webcam using the OpenCV VideoCapture object. It converts the captured image to RGB format and uses the Mediapipe Hands module to detect the user's hand landmarks.

If a hand is detected, the program checks if any of the buttons on the virtual keyboard are being hovered over by the hand. If a button is being hovered over, the program highlights the button by drawing a rectangle around it. The program then calculates the distance between the tip of the index finger and the tip of the middle finger. If the distance is less than a certain threshold, the program simulates a key press by sending the corresponding key to the virtual keyboard.

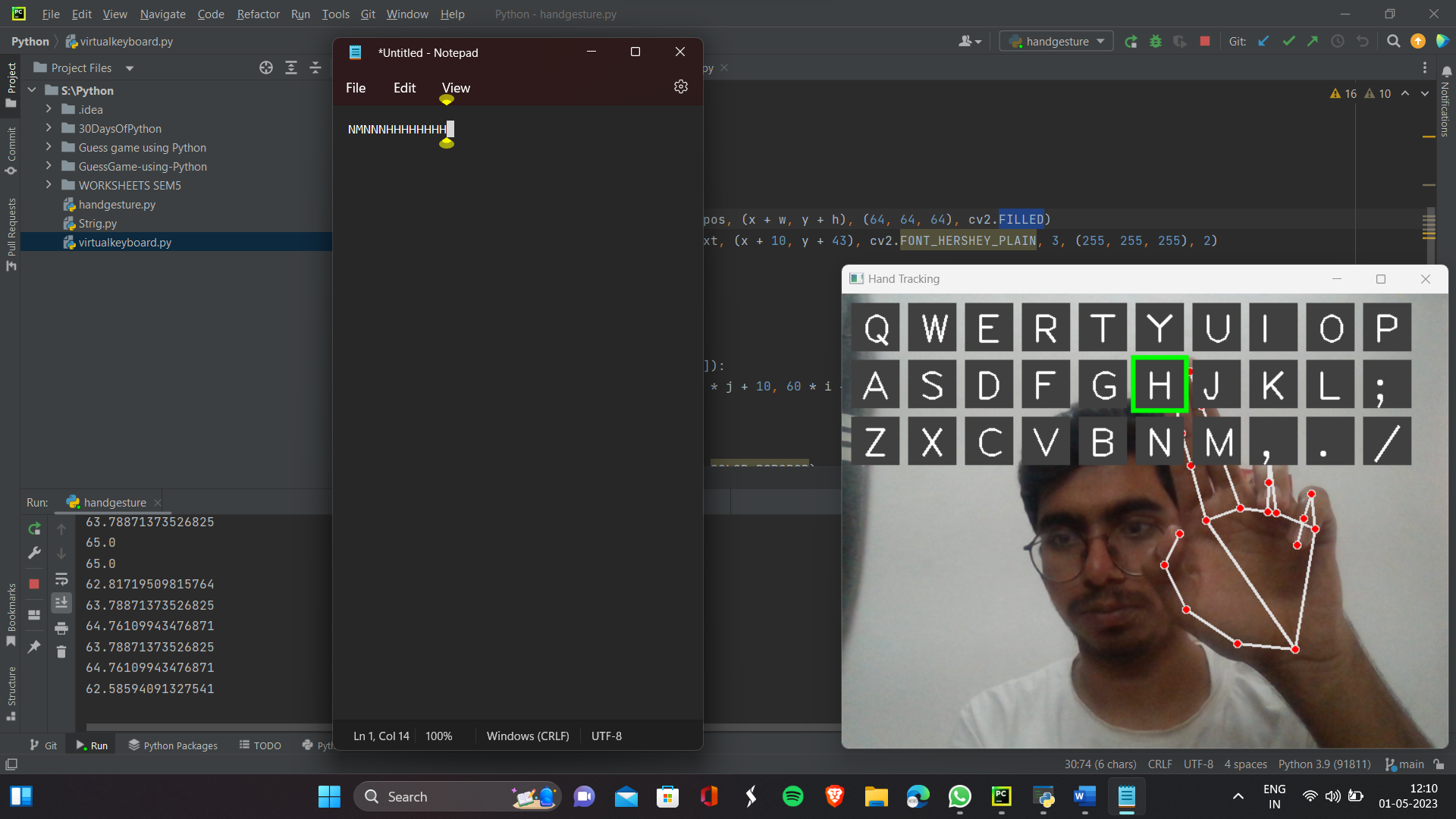
The program then draws the virtual keyboard on the captured image using the draw() function, and displays the resulting image in a window using the OpenCV imshow() function. The program exits the loop and closes the window when the user presses the 'Q' key on their physical keyboard.

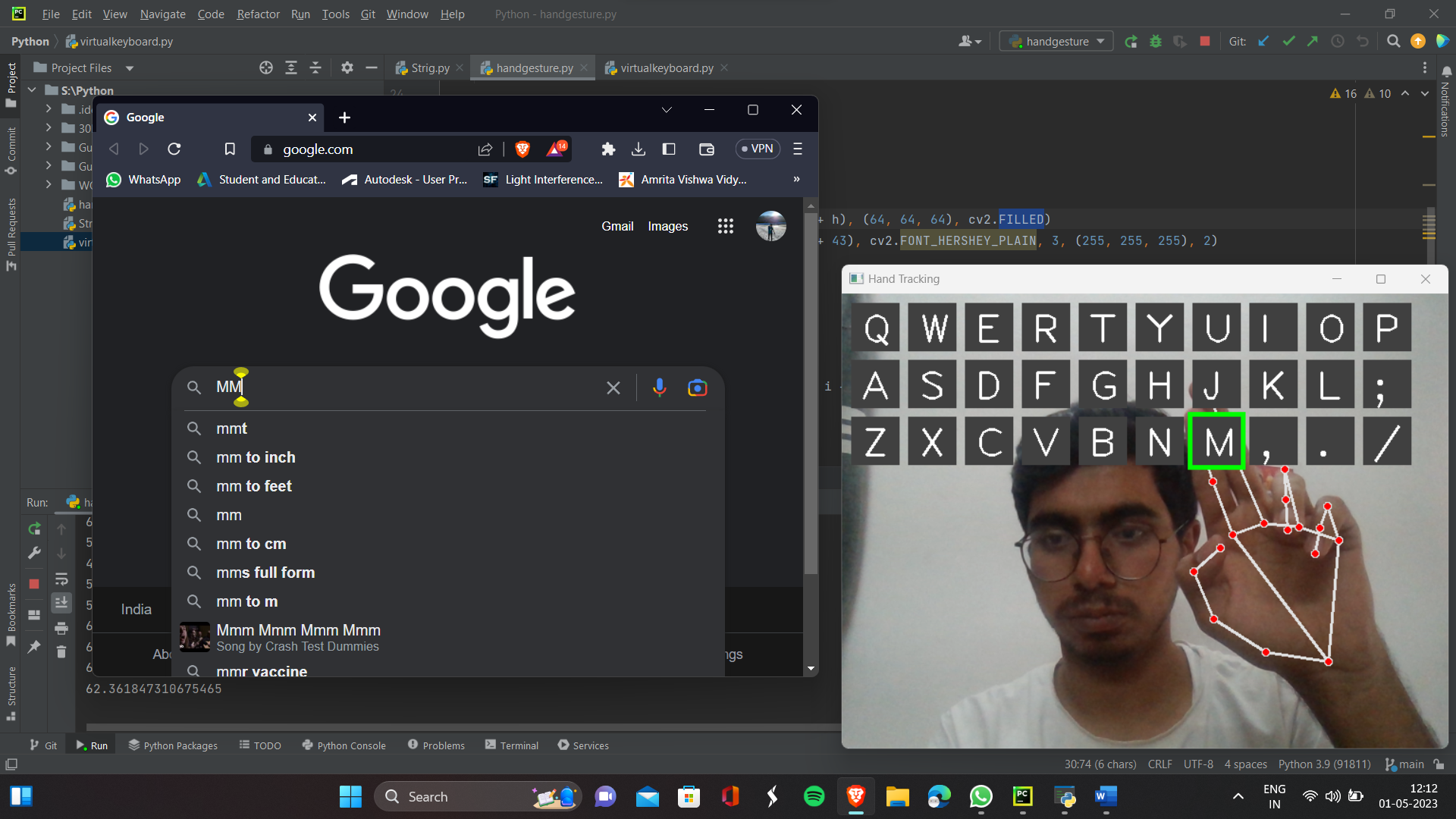
1. Import necessary libraries:
   * **cv2**: OpenCV library for image and video processing
   * **mediapipe**: Mediapipe library for hand tracking
   * **pynput.keyboard**: Pynput library for controlling keyboard input
   * **time.sleep**: Sleep function to add a delay
   * **math**: Math library for mathematical operations
2. Define the webcam index for capturing the video feed and create a VideoCapture object using OpenCV.
3. Create a Hands object from the Mediapipe Hands module for hand tracking and a DrawingUtils object for drawing the landmarks and connections.
4. Define the layout of the virtual keyboard as a nested list of strings. Each string represents a letter, number, or special character on the keyboard.
5. Create a list of buttons for the virtual keyboard by iterating over the nested list and creating a **Store** object for each button. The **Store** class contains the position, size, and text of each button.
6. Enter a loop where each iteration reads a frame from the webcam using the **cap.read()** function. The loop breaks if the user presses the 'Q' key on their physical keyboard.
7. Convert the captured image to RGB format using the **cv2.cvtColor()** function.
8. Use the Hands object to process the RGB image and detect the landmarks of any hands in the image. If a hand is detected, draw the landmarks and connections on the image using the **mpDraw.draw\_landmarks()** function.
9. Extract the coordinates of the detected hand landmarks and store them in a list called **lmList**.
10. For each button in the virtual keyboard, check if the coordinates of the index finger landmark (**lmList[8]**) are within the boundaries of the button. If so, draw a rectangle around the button to highlight it.
11. Calculate the distance between the index finger tip (**lmList[8]**) and the middle finger tip (**lmList[12]**) using the **math.hypot()** function. If the distance is less than a certain threshold, simulate a key press by sending the corresponding key to the virtual keyboard using the **keyboard.press()** function from the Pynput library.
12. Draw the virtual keyboard on the captured image using the **draw()** function, which draws rectangles and text for each button.
13. Display the resulting image in a window using the **cv2.imshow()** function.
14. Repeat steps 6-13 for each frame of the video feed, until the user presses the 'Q' key to break out of the loop.
15. Release the resources used by the VideoCapture object and close the window using the **cap.release()** and **cv2.destroyAllWindows()** functions, respectively.

Demonstration of Virtual Keyboard:



When Using Notepad:





# Virtual Mouse

##### This is a Python code for implementing a hand tracking mouse controller. It uses OpenCV to capture the video stream and detect the hand using a hand tracking module. The program then checks the position of the index and middle fingers to determine whether the user is moving or clicking the mouse.

##### In the moving mode, the program uses the position of the index finger to control the movement of the mouse cursor. The x and y coordinates of the finger are converted to the screen coordinates and then smoothed using a simple smoothing function. The smoothed coordinates are then used to move the mouse cursor using the Autopy library.

##### In the clicking mode, the program checks the distance between the index and middle fingers. If the distance is short, the program performs a left-click using the Autopy library.

##### The program also displays the current frame rate on the screen.

Step 1: Import Required Libraries and Modules

The first few lines of code import the required libraries and modules, such as OpenCV, NumPy, time, and Autopy. The most important module here is "handtracking", which is stored in a separate file called "handtracking.ipynb". This module contains a class called "handDetector" that is used to detect the hand and fingers.

Step 2: Set Constants and Variables

The next few lines of code set some constants and variables that are used throughout the code. These include the width and height of the camera feed, the frame reduction value, and the smoothing factor for the mouse cursor movement.

Step 3: Initialize Variables

The next lines of code initialize some variables such as the previous and current locations of the mouse cursor, the previous time, and the capture object for the video stream.

Step 4: Configure the Camera

This step configures the camera by setting its width and height using the "cap.set()" method.

Step 5: Initialize the Hand Detector Object

The next line of code initializes the hand detector object by creating an instance of the "handDetector" class defined in the "handtracking" module.

Step 6: Get the Screen Size

The next line of code gets the size of the user's screen using the "autopy.screen.size()" method.

Step 7: Start the Main Loop

The main loop starts here, which will run until the user stops the program.

Step 8: Read the Camera Feed

In each iteration of the loop, the program reads a frame from the camera feed using the "cap.read()" method.

Step 9: Find the Hand and Fingers

The program then passes the frame to the hand detector object's "findHands()" method to detect the hand and fingers. This method returns the modified frame with the hand landmarks and the list of landmarks.

Step 10: Check Which Fingers are Up

The program then checks which fingers are up by calling the hand detector object's "fingersUp()" method. This method returns a list of boolean values that represent whether each finger is up or down.

Step 11: Moving Mode - Control the Mouse Cursor

If only the index finger is up, the program enters the moving mode. In this mode, the program converts the x and y coordinates of the index finger to screen coordinates using the "np.interp()" method. The program then smoothes the coordinates using a simple smoothing function that calculates the average of the previous and current coordinates. The smoothed coordinates are then used to move the mouse cursor using the "autopy.mouse.move()" method.

Step 12: Clicking Mode - Perform a Left-Click

If both the index and middle fingers are up, the program enters the clicking mode. In this mode, the program checks the distance between the index and middle fingers using the "detector.findDistance()" method. If the distance is less than a certain threshold, the program performs a left-click using the "autopy.mouse.click()" method.

Step 13: Display the Frame Rate

The program then calculates and displays the frame rate using the current time and the previous time.

Step 14: Display the Modified Frame

Finally, the program displays the modified frame on the screen using the "cv2.imshow()" method and waits for a key press using the "cv2.waitKey()" method.

That's it! This program is a simple but effective implementation of a hand tracking mouse controller using OpenCV and Autopy.

These lines of code initialize three variables used throughout the program:

wCam and hCam set the width and height of the camera capture window to 640 and 480 pixels respectively.

frameR is used to define the size of the region of interest (ROI) rectangle in the camera capture window. This rectangle is defined by the coordinates (frameR, frameR) in the top-left corner and (wCam - frameR, hCam - frameR) in the bottom-right corner.

smoothening is a value used to smooth the movement of the mouse cursor. Its value is set to 7 arbitrarily.

next lines

These lines of code initialize the following variables:

pTime stores the previous time value used to calculate the frame rate.

plocX and plocY store the previous position of the mouse cursor used for smoothing.

clocX and clocY store the current position of the mouse cursor used for smoothing.

cap is a cv2.VideoCapture object used to capture video frames from the default camera (index 0).

detector is a handDetector object created using the handtracking module. It is used to detect the hand landmarks and fingers.

wScr and hScr store the width and height of the screen, respectively, used to convert the coordinates of the detected hand landmarks to the screen coordinates. autopy.screen.size() is used to get the size of the screen.

while loop

This is the main loop of the program. It starts with Step1 where the program reads a frame from the webcam using `cap.read()` and detects the landmarks on the hand using the `detector.findHands(img)` and `detector.findPosition(img)` methods of the `handDetector` object.

Step2 extracts the position of the tip of the index and middle fingers from the `lmList` if it is not empty.

In Step3, the program checks which fingers are up using the `detector.fingersUp()` method.

Step4 checks if only the index finger is up and the middle finger is down. If true, the program enters Moving Mode where the program moves the mouse pointer according to the position of the index finger on the screen.

Step5 maps the position of the index finger from the camera frame to the screen size using the `np.interp()` method.

Step6 smoothes the movement of the mouse by taking the average of the previous position and the current position of the mouse pointer.

Step7 moves the mouse using the `autopy.mouse.move()` method and updates the previous position of the mouse.

Step8 checks if both index and middle fingers are up. If true, the program enters Clicking Mode.

Step9 finds the distance between the tip of the index and middle fingers using the `detector.findDistance()` method.

Step10 checks if the distance between the index and middle fingers is short, and if true, the program clicks the mouse using the `autopy.mouse.click()` method.

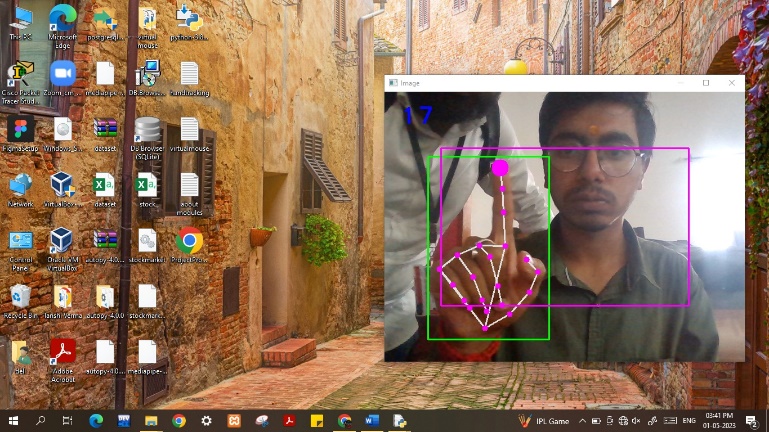
The program also draws a rectangle around the center of the screen, and circles around the index finger in Moving Mode and the position of the fingers in Clicking Mode. It also calculates and displays the frame rate of the video stream.

after while loop

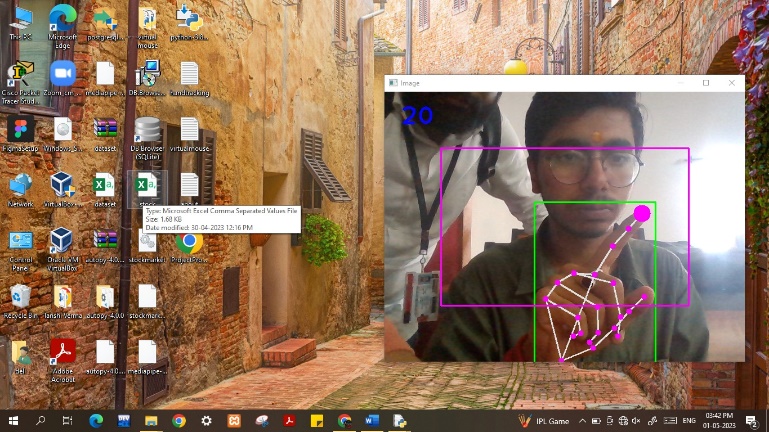
Step 11 calculates the frame rate of the program by measuring the time it takes to process each frame. The current time is stored in the variable `cTime`, and the time taken to process the previous frame is stored in `pTime`. The frame rate is calculated as the reciprocal of the time taken to process the current frame, i.e., `fps = 1/(cTime-pTime)`.

In Step 12, the processed image is displayed using the `cv2.imshow()` function. The first argument to this function is the window name, which is set to "Image". The second argument is the image that is to be displayed. The `cv2.waitKey()` function is used to wait for a specified number of milliseconds (in this case, 1 millisecond) for a keyboard event. If a key is pressed during this time, the corresponding ASCII value is returned. If no key is pressed, the function returns -1. This loop runs indefinitely until the user presses the 'q' key to quit the program.

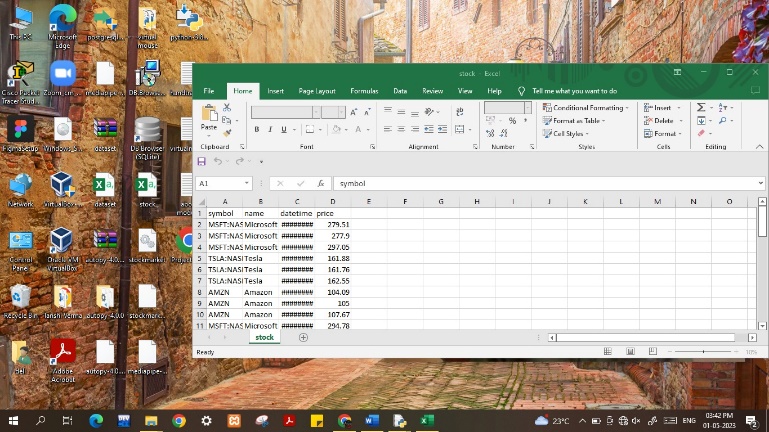
Demonstration of Virtual Mouse:

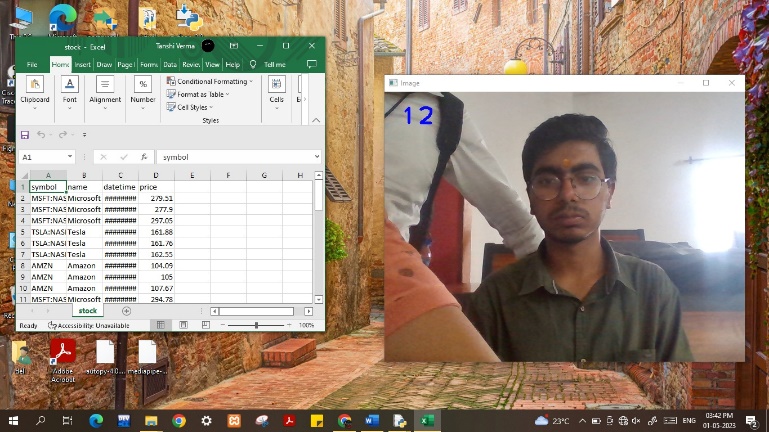


Selecting the Excel File:



Opening of Excel File:





##### References

1. <https://www.nature.com/articles/s41598-022-08133-z>
2. <https://www.semanticscholar.org/paper/Human-computer-interface-using-hand-gesture-based-Jalab-Omer/1c7e28fc4855f1948026354f12d5623918d37dda>
3. <https://www.youtube.com/watch?v=NJmk1DUyyB8>
4. <https://www.youtube.com/watch?v=L6Dbtqi7SQM>
5. <https://www.youtube.com/watch?v=9zqgvsfbz8M>
6. <https://becominghuman.ai/deep-learning-hand-gesture-recognition-b265f4e6cf02>
7. <https://becominghuman.ai/deep-learning-hand-gesture-recognition-b265f4e6cf02>
8. <https://dl.acm.org/doi/10.1145/3293353.3293421>
9. <https://www.nature.com/articles/s41598-022-08133-z>
10. <https://www.mdpi.com/1424-8220/19/24/5429>