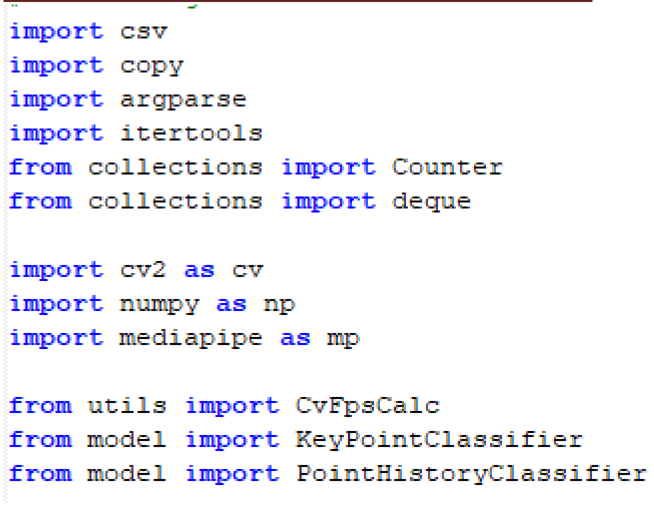
app.py

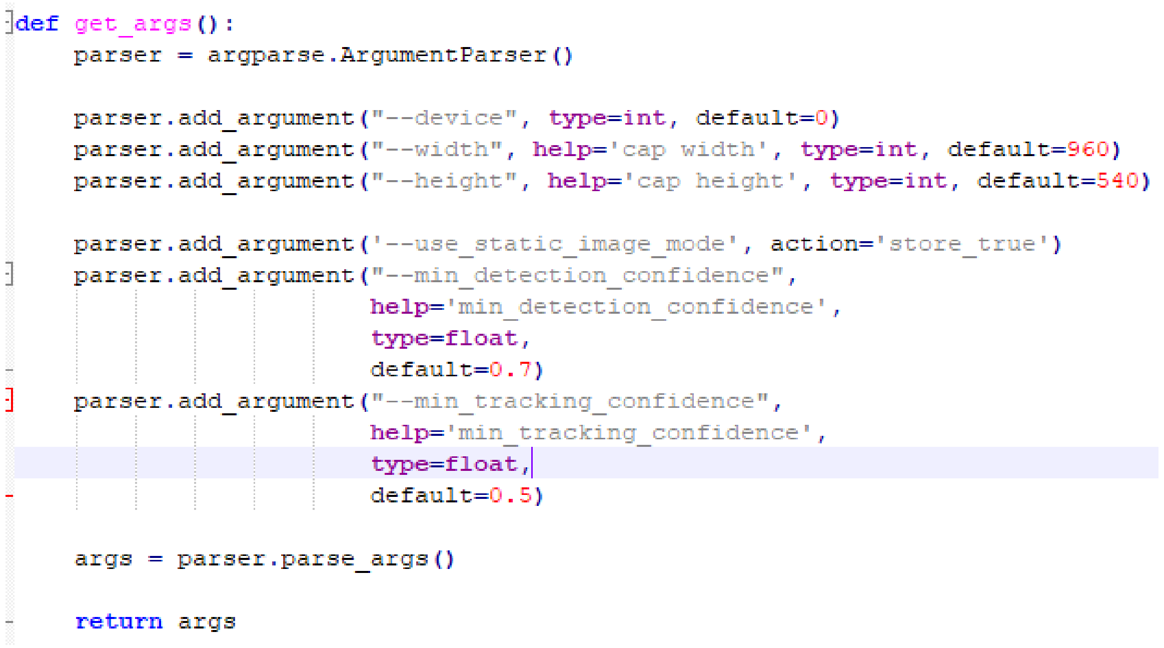
Sunday, August 6, 2023
8:07 PM

IMPORT ALL LIBRARIES REQUIRED



```python
import csv
import copy
import argparse
import itertools
from collections import Counter
from collections import deque

import cv2 as cv
import numpy as np
import mediapipe as mp

from utils import CvFpsCalc
from model import KeyPointClassifier
from model import PointHistoryClassifier
```

Parse command-line arguments using the argparse library



```python
def get_args():
    parser = argparse.ArgumentParser()

    parser.add_argument("--device", type=int, default=0)
    parser.add_argument("--width", help='cap width', type=int, default=960)
    parser.add_argument("--height", help='cap height', type=int, default=540)

    parser.add_argument('--use_static_image_mode', action='store_true')
    parser.add_argument("--min_detection_confidence",
                        help='min_detection_confidence',
                        type=float,
                        default=0.7)
    parser.add_argument("--min_tracking_confidence",
                        help='min_tracking_confidence',
                        type=float,
                        default=0.5)

    args = parser.parse_args()

    return args
```

This function creates an argument parser and sets up various command-line arguments that can be used to customize the behavior of the script. Here's a breakdown of the arguments:

--device: Specifies the webcam device index to be used for capturing video. Default value is 0 (usually the default built-in webcam).

--width: Specifies the width of the captured video frame. Default value is 960 pixels.

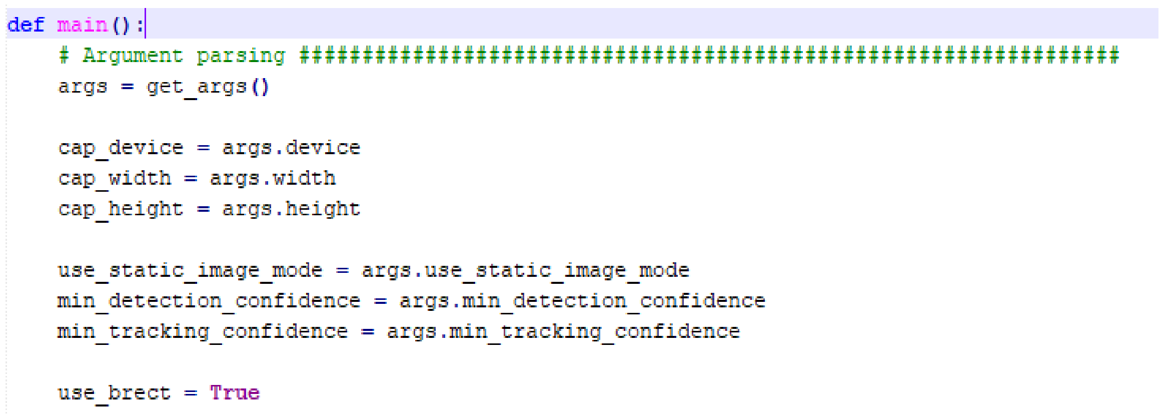
--height: Specifies the height of the captured video frame. Default value is 540 pixels.

--use_static_image_mode: If this flag is present, it enables static image mode in the hand tracking model. This mode is used for processing static images rather than video streams.

--min_detection_confidence: Specifies the minimum confidence value(it's between 0 and 1) required for a hand to be detected by the hand tracking model. Default value is 0.7.

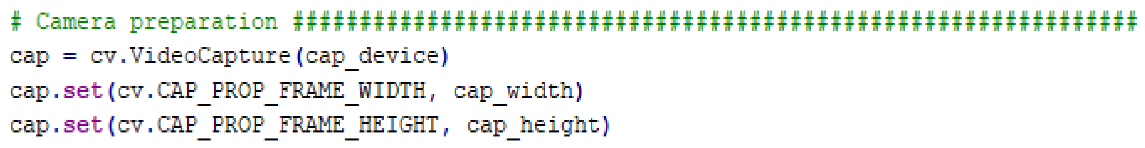
--min_tracking_confidence: Specifies the minimum confidence value required for the hand tracking model to continue tracking a detected hand. Default value is 0.5.

Parsing command-line arguments using the get_args function



```
def main():
    # Argument parsing #################################################################
    args = get_args()

    cap_device = args.device
    cap_width = args.width
    cap_height = args.height

    use_static_image_mode = args.use_static_image_mode
    min_detection_confidence = args.min_detection_confidence
    min_tracking_confidence = args.min_tracking_confidence

    use_brect = True
```

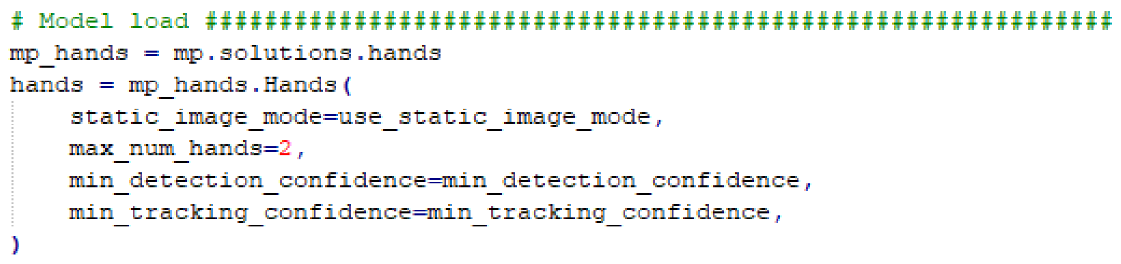
Camera Preparation using computer vision



```
# Camera preparation ###############################################################
cap = cv.VideoCapture(cap_device)
cap.set(cv.CAP_PROP_FRAME_WIDTH, cap_width)
cap.set(cv.CAP_PROP_FRAME_HEIGHT, cap_height)
```

The **OpenCV VideoCapture** object is created to capture video from the specified webcam device. The frame dimensions are set according to the parsed arguments.

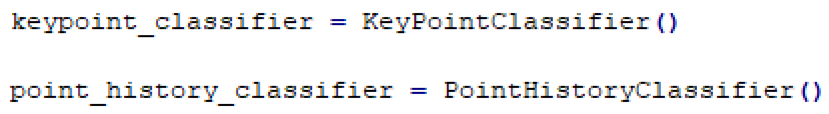
MediaPipe Hands model



```
# Model load #############################################################
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(
    static_image_mode=use_static_image_mode,
    max_num_hands=2,
    min_detection_confidence=min_detection_confidence,
    min_tracking_confidence=min_tracking_confidence,
)
```

Setting up the **mp_hands** variable to refer to the **hands module** within the **mp.solutions namespace** of the MediaPipe library. This module provides **functionality for hand detection and tracking** using MediaPipe's pre-trained models. By using mp_hands, you can access the **Hands class and its methods** to perform hand-related tasks such as detecting and tracking hand landmarks in images or video frames.

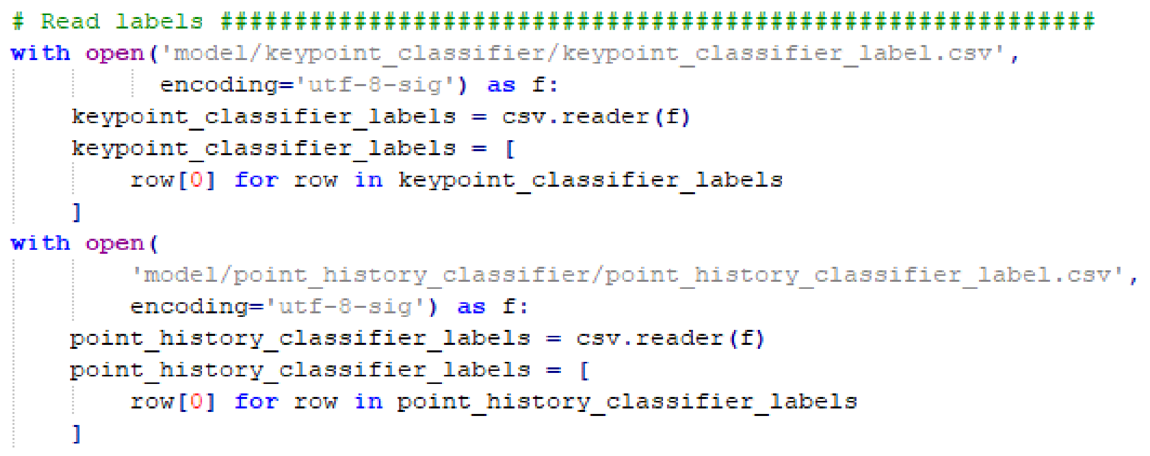
Classifier Initialization



```
keypoint_classifier = KeyPointClassifier()

point_history_classifier = PointHistoryClassifier()
```

Instances of the KeyPointClassifier and PointHistoryClassifier classes are created.

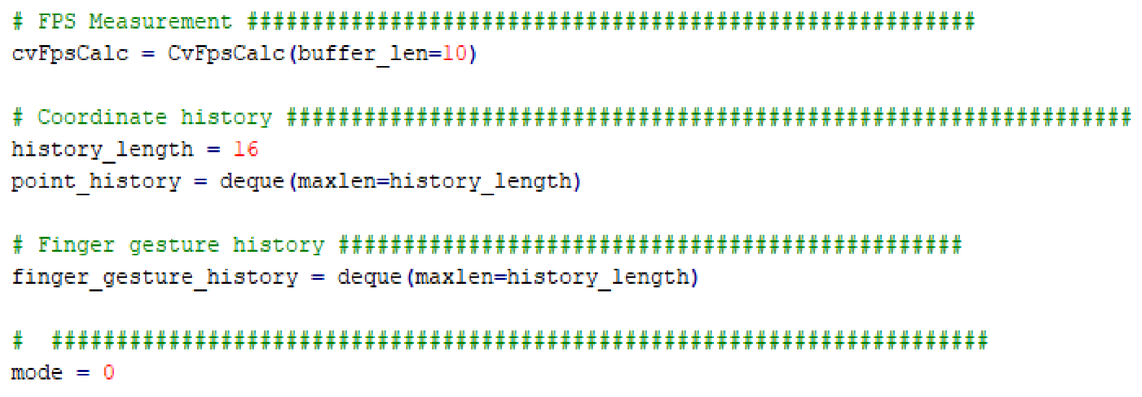
Reading labels from a CSV files (keypoint_classifier_label.csv and point_history_classifier_label.csv)



```
# Read labels ###########################################################
with open('model/keypoint_classifier/keypoint_classifier_label.csv',
          encoding='utf-8-sig') as f:
    keypoint_classifier_labels = csv.reader(f)
    keypoint_classifier_labels = [
        row[0] for row in keypoint_classifier_labels
    ]
with open(
        'model/point_history_classifier/point_history_classifier_label.csv',
        encoding='utf-8-sig') as f:
    point_history_classifier_labels = csv.reader(f)
    point_history_classifier_labels = [
        row[0] for row in point_history_classifier_labels
    ]
```

with open('model/keypoint_classifier/keypoint_classifier_label.csv', encoding='utf-8-sig') as f:: This line opens the specified CSV file (keypoint_classifier_label.csv) for reading. The encoding='utf-8-sig' argument. is used to ensure proper handling of the file's encoding.
The encoding='utf-8-sig' argument is typically used when working with text files in Python to handle encoded in UTF-8 format that might have a byte order mark (BOM) at the beginning
keypoint_classifier_labels = csv.reader(f): This line creates a CSV reader object, keypoint_classifier_labels, which allows you to iterate through the rows of the CSV file.
keypoint_classifier_labels = [row[0] for row in keypoint_classifier_labels]: Here, a list comprehension is used to iterate through each row in the CSV file and extract the value from the first column (row[0]). This effectively creates a list of labels corresponding to the output classes of your keypoint classifier.

Define some Variables



```
# FPS Measurement ########################################################
cvFpsCalc = CvFpsCalc(buffer_len=10)

# Coordinate history #################################################################
history_length = 16
point_history = deque(maxlen=history_length)

# Finger gesture history ################################################
finger_gesture_history = deque(maxlen=history_length)

#  ########################################################################
mode = 0
```

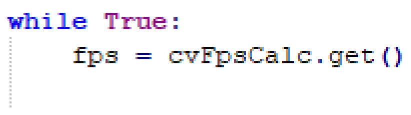
First line initializes an object called cvFpsCalc of the **class CvFpsCalc with a buffer length of 10**. It's used to calculate and maintain the frames per second (FPS) of your video stream.

deque stands for "double-ended queue," and it's a data structure that provides a way to

store and manipulate elements in a collection. It's a generalization of stacks and queues, allowing you to add or remove elements from both ends efficiently.
In Python, the collections module provides the deque class.

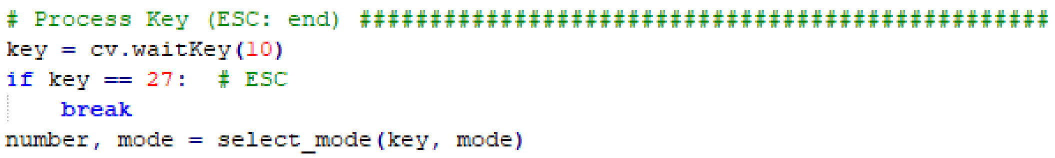
This initializes the mode variable to 0. This variable is used to control the mode of your application.

Main loop and calculation the current frames per second



```
while True:
    fps = cvFpsCalc.get()
```

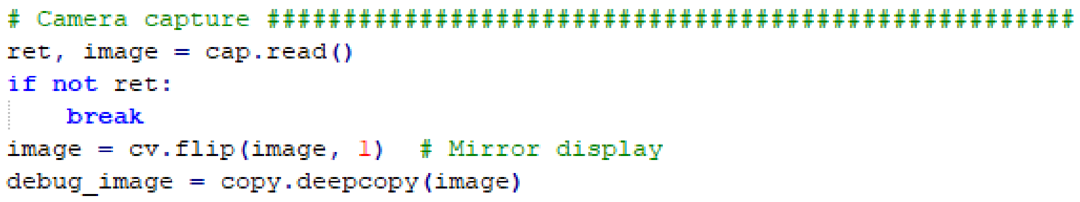
Processing key inputs



```
# Process Key (ESC: end) #################################################
key = cv.waitKey(10)
if key == 27:  # ESC
    break
number, mode = select_mode(key, mode)
```

select_mode is a function used to give a number (between 0 & 9) and mode depending on pressed that is described in Self-Written Functions portion.

Camera Capturing

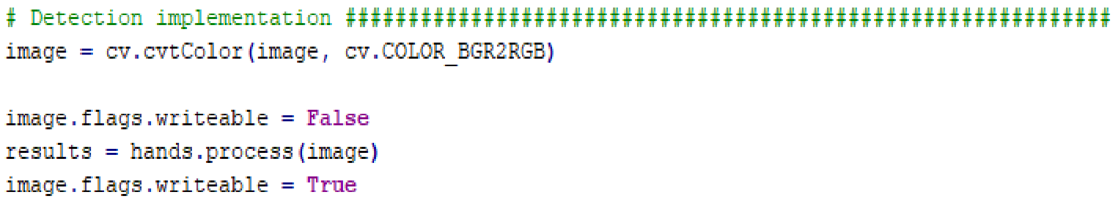


```
# Camera capture #####################################################
ret, image = cap.read()
if not ret:
    break
image = cv.flip(image, 1)  # Mirror display
debug_image = copy.deepcopy(image)
```

If the read is successful (ret is True), it proceeds to flip the image horizontally using cv.flip() for a mirrored effect. The debug_image variable is then assigned a deep copy of the modified image.

deepcopy is a concept and a method used in programming, particularly in Python, to create a new copy of an object that is a completely independent duplicate.

Image Conversion and detect hand related features in image

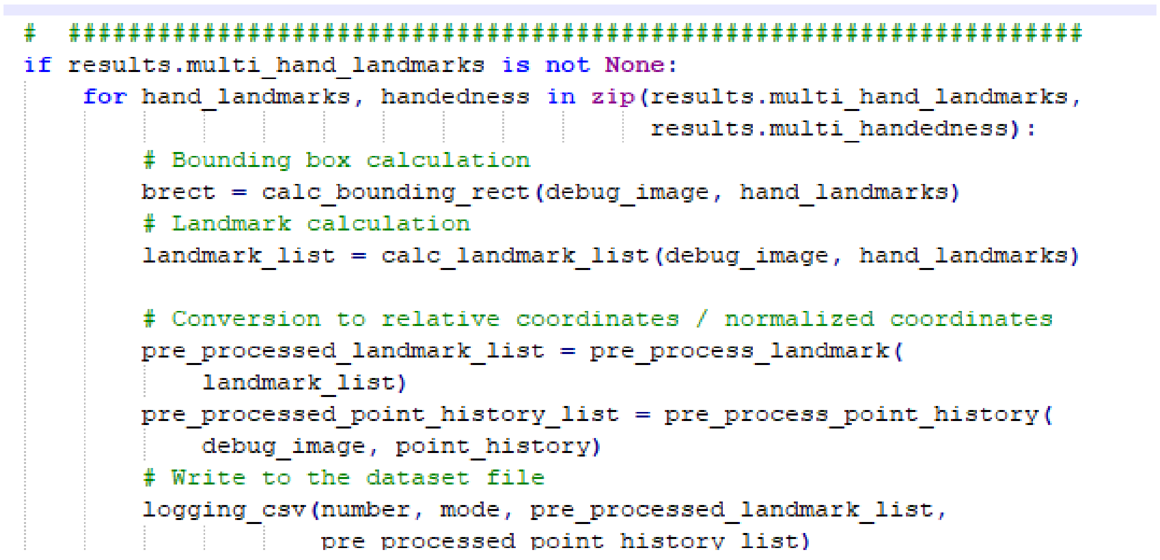


```
# Detection implementation #############################################################
image = cv.cvtColor(image, cv.COLOR_BGR2RGB)

image.flags.writeable = False
results = hands.process(image)
image.flags.writeable = True
```

In this code snippet, the captured image is converted from the BGR color space to the RGB color using **cv.cvtColor()**. The **writeable attribute** of the image is temporarily set to False to optimize memory usage. The **hands.process()** function is applied to the image to detect hands or hand-related features. Finally, the writeable attribute is set back to True, allowing modifications to the image.

The **main difference between BGR and RGB** lies in the order of color channels used to represent an image's colors. BGR stands for Blue, Green, Red, while RGB stands for Red, Green, Blue. Both BGR and RGB are color models used to represent colors in digital images, but they use different orders for the color channels.

Detecting landmarks, Preprocessing and saving into CSV Files

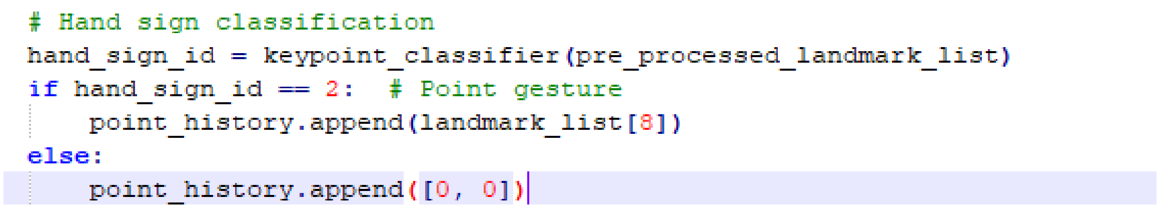


```
#  ####################################################################
if results.multi_hand_landmarks is not None:
    for hand_landmarks, handedness in zip(results.multi_hand_landmarks,
                                          results.multi_handedness):
        # Bounding box calculation
        brect = calc_bounding_rect(debug_image, hand_landmarks)
        # Landmark calculation
        landmark_list = calc_landmark_list(debug_image, hand_landmarks)

        # Conversion to relative coordinates / normalized coordinates
        pre_processed_landmark_list = pre_process_landmark(
            landmark_list)
        pre_processed_point_history_list = pre_process_point_history(
            debug_image, point_history)
        # Write to the dataset file
        logging_csv(number, mode, pre_processed_landmark_list,
                    pre_processed_point_history_list)
```

results.multi_hand_landmarks: This represents the detected landmarks for each hand in the current frame. If there are hands detected, the code enters the loop to process each detected hand.
for hand_landmarks, handedness in zip(results.multi_hand_landmarks, results.multi_handedness): This loop iterates through the detected hand landmarks and corresponding handedness (left or right hand).
brect = calc_bounding_rect(debug_image, hand_landmarks): This line calculates the bounding rectangle (brect) around the detected hand landmarks. There is the **calc_bounding_rect function** calculates the minimum bounding rectangle that encloses the hand landmarks.
landmark_list = calc_landmark_list(debug_image, hand_landmarks): This line calculates a list list of individual landmark coordinates for the detected hand using the calc_landmark_list function. These landmarks represent specific points on the hand, such as fingertips and joints.
pre_processed_landmark_list = pre_process_landmark(landmark_list): The pre_process_landmark function is applied to the calculated landmark list. This function likely performs some preprocessing steps on the landmark coordinates before they are used for further analysis.
pre_processed_point_history_list = pre_process_point_history(debug_image, point_history): The pre_process_point_history function seems to preprocess the historical point history stored in the point_history deque. This could involve transforming the historical data in some way for analysis or visualization.
logging_csv(number, mode, pre_processed_landmark_list, pre_processed_point_history_list):

This line calls a function named logging_csv and passes several pieces of information including number, mode, pre-processed landmark list, and pre-processed point history list. This function likely logs or records this information to a CSV file for later analysis or dataset creation.

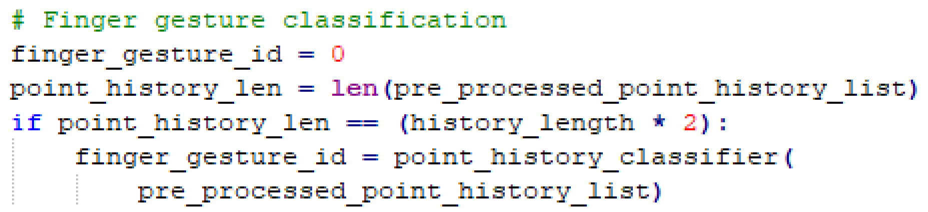
Hand Sign Classification (keypoint_classifier)



```
# Hand sign classification
hand_sign_id = keypoint_classifier(pre_processed_landmark_list)
if hand_sign_id == 2:  # Point gesture
    point_history.append(landmark_list[8])
else:
    point_history.append([0, 0])
```

- The processed landmark data **pre_processed_landmark_list** is fed into the **keypoint_classifier function**, which is a classifier model designed to classify hand signs or gestures.
- The resulting **hand_sign_id** represents the predicted hand gesture label.
- If the predicted gesture is identified as a "**point gesture**" (hand_sign_id is equal to 2), then code appends the **landmark coordinates of the 8th point (presumably a fingertip)** from the landmark list to the **point_history deque**. This could be used to track the movement of the fingertip over time.
- If the predicted gesture is not a point gesture, the code **appends [0, 0] to the point_history**. This will serve as a placeholder for non-point gestures.

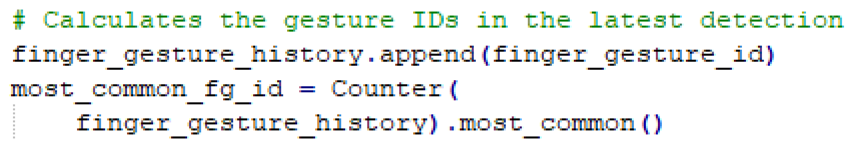
Finger Gesture Classification (point_history_classifier):



```
# Finger gesture classification
finger_gesture_id = 0
point_history_len = len(pre_processed_point_history_list)
if point_history_len == (history_length * 2):
    finger_gesture_id = point_history_classifier(
        pre_processed_point_history_list)
```

- **finger_gesture_id** is initialized to 0.
- **point_history_len** is calculated as the length of the pre-processed point history list (pre_processed_point_history_list).
- If the length of the point history matches twice the history_length(16), it indicates that **historical data** has been accumulated.
- In that case, the point_history_classifier function is used to classify a finger gesture based the historical point data stored in **pre_processed_point_history_list**. The result is assigned finger_gesture_id

Calculate the most common Gesture ID



```
# Calculates the gesture IDs in the latest detection
finger_gesture_history.append(finger_gesture_id)
most_common_fg_id = Counter(
    finger_gesture_history).most_common()
```

Appending to Finger Gesture History (finger_gesture_history):

- After classifying the finger gesture using point_history_classifier, the calculated

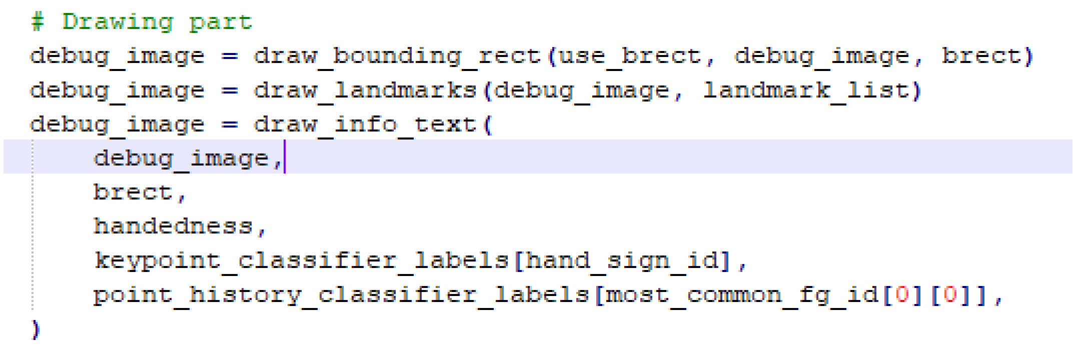
finger_gesture_id is appended to the finger_gesture_history deque.
- The purpose of this history is to keep track of previously classified finger gesture IDs.

Calculating the Most Common Gesture ID:

- The **Counter class** from the collections module is used to create a counter object from the finger_gesture_history deque.
- The **most_common()** method of the counter object is called to retrieve a list of tuples, where each tuple contains a finger gesture ID and its corresponding count in the history.
- The variable **most_common_fg_id** stores this list of tuples.

At this point, most_common_fg_id contains a list of tuples where each tuple contains a gesture ID and the count of its occurrences in the history. The gesture ID with the **highest** (i.e., the most common gesture) can be accessed using **most_common_fg_id[0][0].**

Drawing rectangle, landmarks and text



```
# Drawing part
debug_image = draw_bounding_rect(use_brect, debug_image, brect)
debug_image = draw_landmarks(debug_image, landmark_list)
debug_image = draw_info_text(
        debug_image,
        brect,
        handedness,
        keypoint_classifier_labels[hand_sign_id],
        point_history_classifier_labels[most_common_fg_id[0][0]],
)
```

Drawing Bounding Rectangle:

- The function draw_bounding_rect is used to draw a bounding rectangle around the detected hand region.
- The boolean variable use_brect determines whether this rectangle should be drawn. based on the value of use_brect from earlier in the code.
- The function takes the debug_image, the use_brect flag, and the calculated bounding rectangle brect as arguments. The modified debug image is returned.

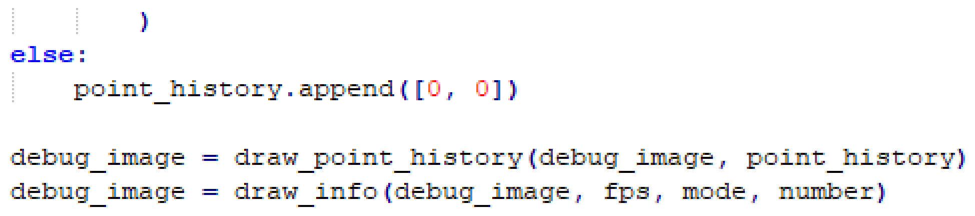
Drawing Landmarks:

- The function draw_landmarks is used to draw landmarks on the hand detected in the image.
- It takes the debug_image and the list of landmark_list as arguments and returns the modified debug image with landmarks drawn.

Drawing Information Text:

- The function draw_info_text is used to overlay information text on the debug image.

Drawing point_history, mode, number and fps



```
            )
else:
    point_history.append([0, 0])

debug_image = draw_point_history(debug_image, point_history)
debug_image = draw_info(debug_image, fps, mode, number)
```

This else is corresponding to **if results.multi_hand_landmarks is not None**

Drawing Point History:

- The function draw_point_history is used to draw the history of tracked points on the debug image.

Drawing Additional Information:

- The function draw_info is used to overlay additional information on the debug image, including the current frames per second (fps), the current mode, and the number.

Show resulted thing and close all windows after escape

```
        # Screen reflection #############################################################
        cv.imshow('Hand Gesture Recognition', debug_image)

    cap.release()
    cv.destroyAllWindows()
```

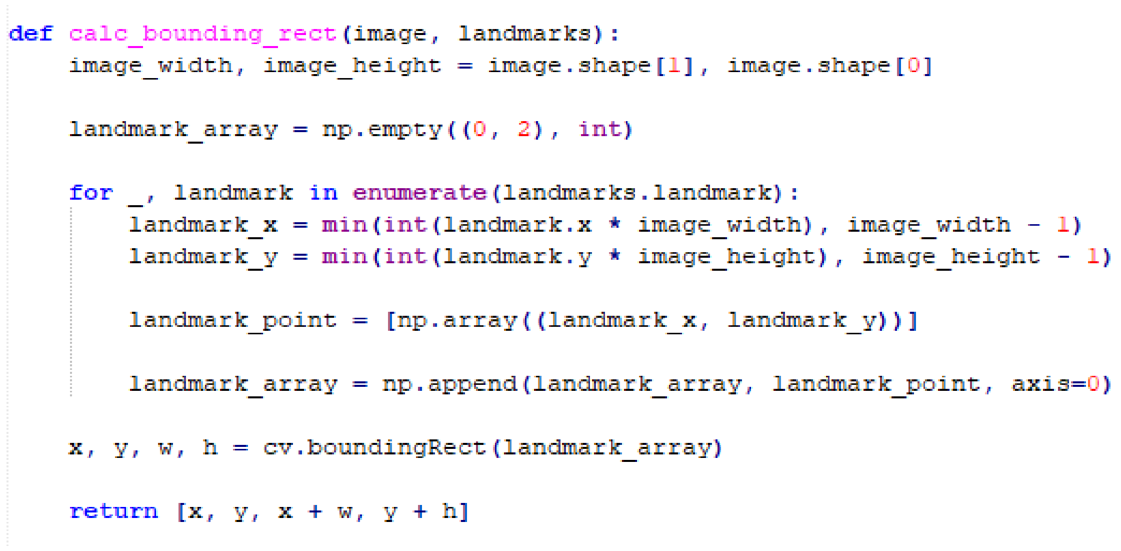
Self-Written Functions

select_mode

```
def select_mode(key, mode):
    number = -1
    if 48 <= key <= 57:  # 0 ~ 9
        number = key - 48
    if key == 110:  # n
        mode = 0
    if key == 107:  # k
        mode = 1
    if key == 104:  # h
        mode = 2
    return number, mode
```

Return number between 0 and 9 and give mode corresponding to specific key

Calculate Bounding Rectangle



```
def calc_bounding_rect(image, landmarks):
    image_width, image_height = image.shape[1], image.shape[0]

    landmark_array = np.empty((0, 2), int)

    for _, landmark in enumerate(landmarks.landmark):
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)

        landmark_point = [np.array((landmark_x, landmark_y))]

        landmark_array = np.append(landmark_array, landmark_point, axis=0)

    x, y, w, h = cv.boundingRect(landmark_array)

    return [x, y, x + w, y + h]
```

image_width, image_height = image.shape[1], image.shape[0]:
- The dimensions of the input image are extracted using the shape attribute of the image.
- image.shape[1] gives the width of the image, and image.shape[0] gives the height.

landmark_array = np.empty((0, 2), int):
- An empty NumPy array is created to store the landmarks' coordinates.

Loop over each landmark in the provided landmarks:
- **for _, landmark in enumerate(landmarks.landmark):**
  enumerate(landmarks.landmark) is used to iterate over the landmarks while also keeping track their index. The enumerate() function is used to add a counter to an iterable and returns pairs the index and item. In this case, it's used to get both the index (which is ignored with _) and the landmark itself.

  Calculate the landmark's pixel coordinates:
  - The coordinates are scaled by the image dimensions to get the corresponding coordinates within the image.
  - **landmark_x = min(int(landmark.x * image_width), image_width - 1):**
    - The x-coordinate of the landmark, scaled to the image width, ensuring it doesn't exceed the image boundaries.
  - **landmark_y = min(int(landmark.y * image_height), image_height - 1):**
    - The y-coordinate of the landmark, scaled to the image height, ensuring it doesn't exceed the image boundaries.

  Store the landmark coordinates in landmark_array:
  - A single landmark's pixel coordinates are stored in an array and appended to landmark_array.

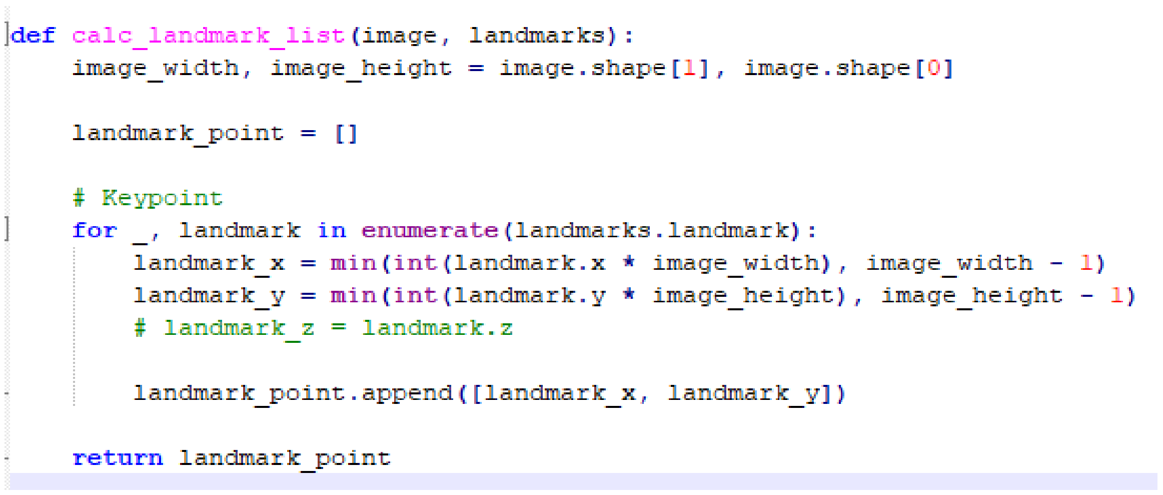
  Calculate the bounding rectangle:
  - x, y, w, h = cv.boundingRect(landmark_array):
  - The **cv.boundingRect()** function calculates the minimal bounding rectangle for a set of points.
  - **x and y** are the coordinates of the top-left corner of the bounding rectangle.
  - **w** is the width of the bounding rectangle, and **h** is the height.

Return the bounding rectangle coordinates:
  - The function returns the coordinates of the **top-left** and **bottom-right** corners of the bounding rectangle, **[x, y, x + w, y + h]**

Pixel coordinates refer to the location of a point within an image, using the row and

column indices of the pixels that make up the image. In a digital image, each pixel is assigned a unique position defined by its horizontal (column) and vertical (row) coordinates.
In most cases, the top-left corner of the image is considered the origin (0,0) of the system. The x-coordinate (column index) increases as you move to the right, and the y-coordinate (row index) increases as you move downward. So, pixel coordinates are used to specify the exact location of a point within the image by providing its row and column

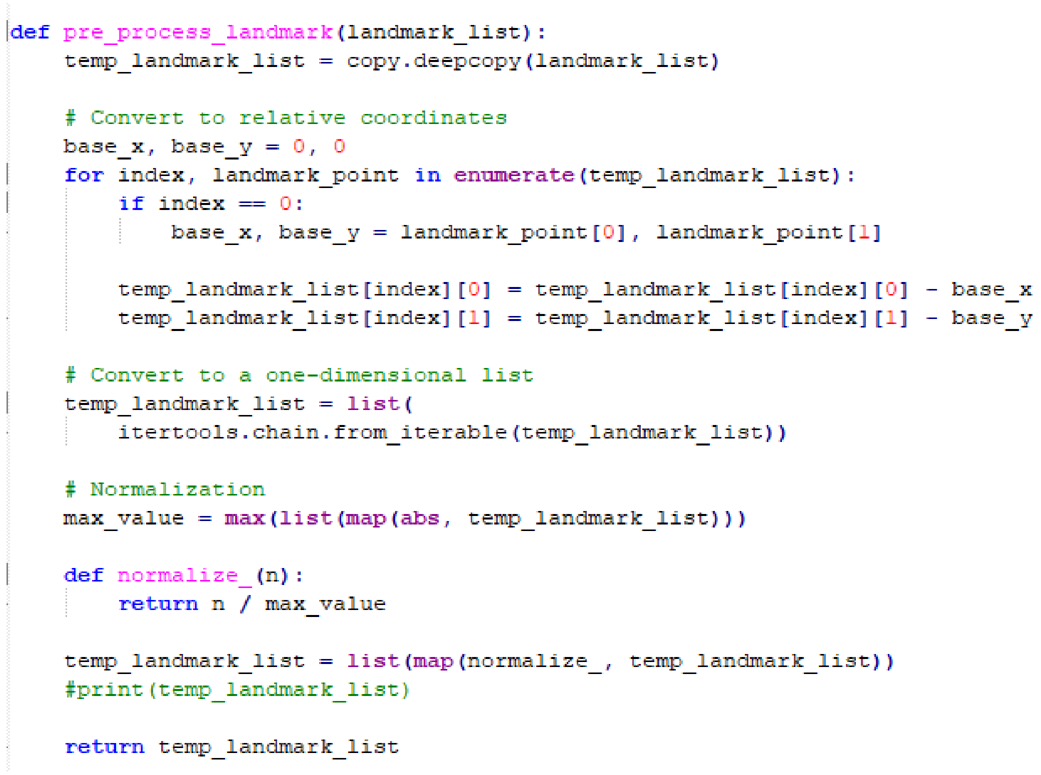
Calculate Landmark list



```
def calc_landmark_list(image, landmarks):
    image_width, image_height = image.shape[1], image.shape[0]

    landmark_point = []

    # Keypoint
    for _, landmark in enumerate(landmarks.landmark):
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)
        # landmark_z = landmark.z

        landmark_point.append([landmark_x, landmark_y])

    return landmark_point
```

- It receives the **image (a numpy array representing the image)** and **landmarks (detected landmarks from mediapipe)**.
- It obtains the width and height of the image using **image.shape[1]** and **image.shape[0]**, respectively.
- It initializes an **empty list called landmark_point to store the pixel coordinates** of each detected landmark.
- The function then iterates over each detected landmark using **enumerate(landmarks.landmark)**. The enumerate function provides both the index of the landmark and the landmark itself.
- For each landmark, it calculates the landmark_x and landmark_y by multiplying the x and y coordinates **(landmark.x and landmark.y)** with the image_width and image_height, respectively. This effectively converts the **coordinates to pixel coordinates within the image's dimensions**. The use of min ensures that the calculated coordinates don't exceed the image dimensions.
- It appends the calculated [landmark_x, landmark_y] to the landmark_point list.

Pre-Processing Landmarks



```python
def pre_process_landmark(landmark_list):
    temp_landmark_list = copy.deepcopy(landmark_list)

    # Convert to relative coordinates
    base_x, base_y = 0, 0
    for index, landmark_point in enumerate(temp_landmark_list):
        if index == 0:
            base_x, base_y = landmark_point[0], landmark_point[1]

        temp_landmark_list[index][0] = temp_landmark_list[index][0] - base_x
        temp_landmark_list[index][1] = temp_landmark_list[index][1] - base_y

    # Convert to a one-dimensional list
    temp_landmark_list = list(
        itertools.chain.from_iterable(temp_landmark_list))

    # Normalization
    max_value = max(list(map(abs, temp_landmark_list)))

    def normalize_(n):
        return n / max_value

    temp_landmark_list = list(map(normalize_, temp_landmark_list))
    #print(temp_landmark_list)

    return temp_landmark_list
```

Deep Copy: The original landmark_list is deep copied to avoid modifying the original data during preprocessing.
Relative Coordinates: The function calculates the base (reference) point using the **first landmark's coordinates (base_x, base_y)**. Then, it subtracts these base coordinates from all other landmark points. This step essentially shifts the coordinate system so that the first landmark becomes the new origin.
Flattening: The 2D list of landmark points is **flattened into a one-dimensional list** using the itertools.chain.from_iterable function. This step transforms the list of lists into a single list of coordinates.
Normalization: The function calculates the maximum absolute value of all coordinates in the flattened list (max_value). Then, it defines a normalize_ function to divide each coordinate by the max_value, effectively normalizing them **to a range between -1 and 1**.
Normalization Applied: The normalize_ function is applied to each coordinate in the flattened list, resulting in a normalized list of coordinates.

Pre-processing Point history

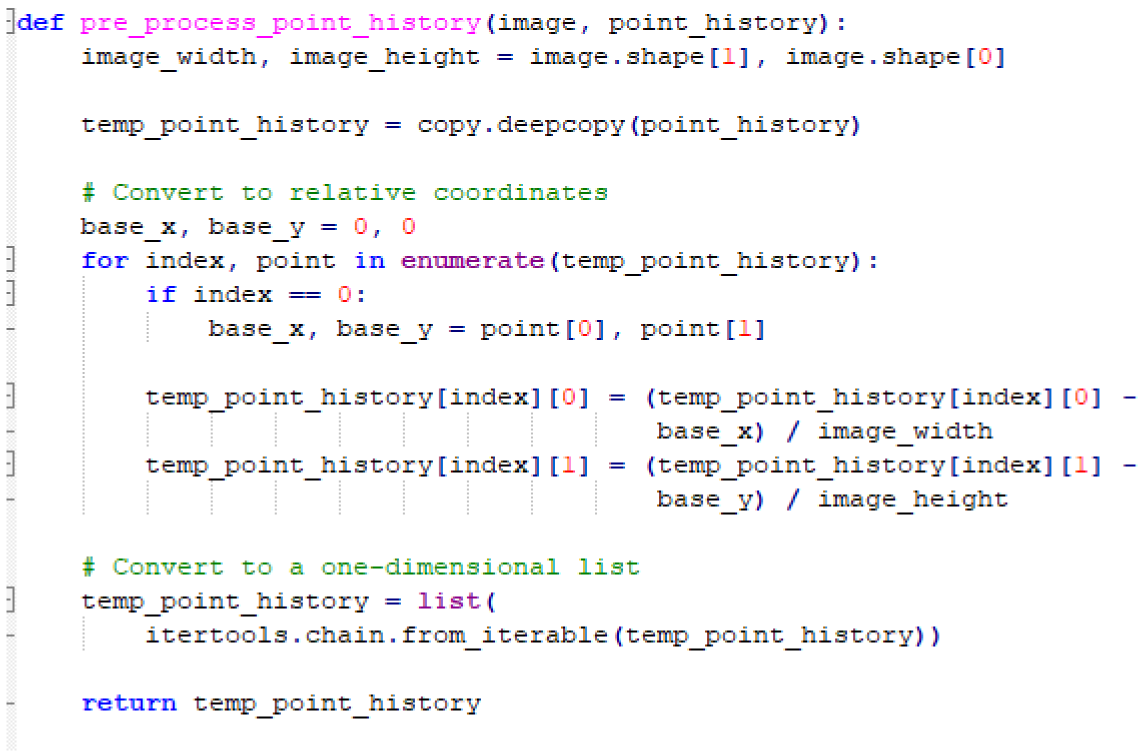
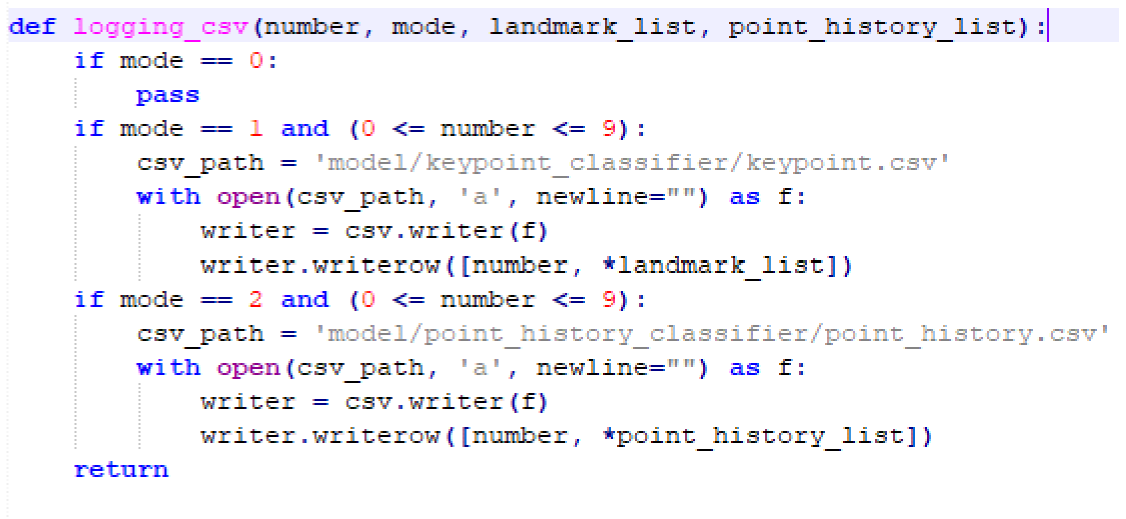

```
def pre_process_point_history(image, point_history):
    image_width, image_height = image.shape[1], image.shape[0]

    temp_point_history = copy.deepcopy(point_history)

    # Convert to relative coordinates
    base_x, base_y = 0, 0
    for index, point in enumerate(temp_point_history):
        if index == 0:
            base_x, base_y = point[0], point[1]

        temp_point_history[index][0] = (temp_point_history[index][0] -
                                        base_x) / image_width
        temp_point_history[index][1] = (temp_point_history[index][1] -
                                        base_y) / image_height

    # Convert to a one-dimensional list
    temp_point_history = list(
        itertools.chain.from_iterable(temp_point_history))

    return temp_point_history
```

Image Dimensions: The function obtains the width and height of the input image.
Deep Copy: The point_history list is deep copied to avoid modifying the original data.
Relative Coordinates: The function calculates the base (reference) point using the first point's coordinates (base_x, base_y). Then, it subtracts the base coordinates from all other points and normalizes them by dividing by the image width and height. This step normalizes the point coordinates based on the dimensions of the image.
Flattening: Similar to the previous function, the 2D list of point history is flattened into a one-dimensional list.
Output: The final output of this function is a list of normalized and flattened point coordinates, which can be used for further analysis or as input to machine learning models.

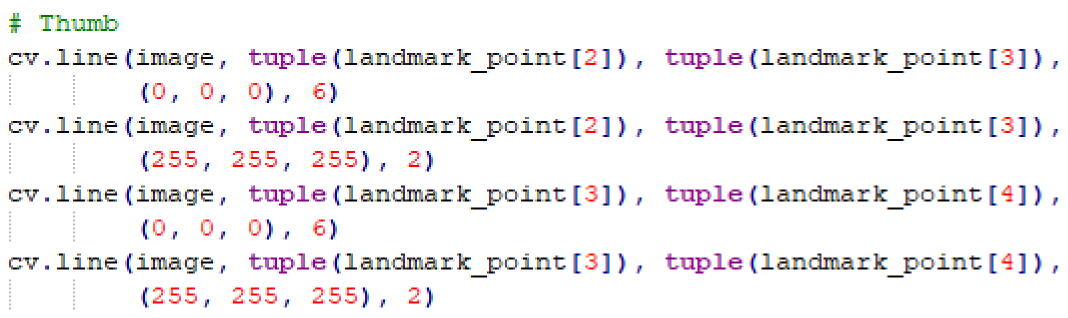
Logging Data into CSV files


```
def logging_csv(number, mode, landmark_list, point_history_list):
    if mode == 0:
        pass
    if mode == 1 and (0 <= number <= 9):
        csv_path = 'model/keypoint_classifier/keypoint.csv'
        with open(csv_path, 'a', newline="") as f:
            writer = csv.writer(f)
            writer.writerow([number, *landmark_list])
    if mode == 2 and (0 <= number <= 9):
        csv_path = 'model/point_history_classifier/point_history.csv'
        with open(csv_path, 'a', newline="") as f:
            writer = csv.writer(f)
            writer.writerow([number, *point_history_list])
    return
```

Arguments: The function takes four arguments: number, mode, landmark_list, and

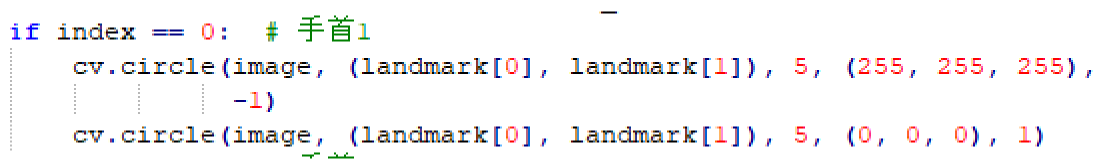
point_history_list.
Mode 0: If mode is 0, the function does nothing. This might indicate that no logging is required in this case.
Mode 1: If mode is 1 and the number is within the range of 0 to 9, the function logs the data from the landmark_list into a CSV file associated with the keypoint classifier model. The data includes the number followed by the values in the landmark_list.
Mode 2: If mode is 2 and the number is within the range of 0 to 9, the function logs the data from the point_history_list into a CSV file associated with the point history classifier model. The data includes the number followed by the values in the point_history_list.

Drawing landmarks on hand



```
# Thumb
cv.line(image, tuple(landmark_point[2]), tuple(landmark_point[3]),
        (0, 0, 0), 6)
cv.line(image, tuple(landmark_point[2]), tuple(landmark_point[3]),
        (255, 255, 255), 2)
cv.line(image, tuple(landmark_point[3]), tuple(landmark_point[4]),
        (0, 0, 0), 6)
cv.line(image, tuple(landmark_point[3]), tuple(landmark_point[4]),
        (255, 255, 255), 2)
```

- **# Thumb:** The code draws lines connecting specific landmarks that represent the thumb. The thumb consists of three landmarks: **one for the base, one for the first joint, and one for the fingertip**.
- For each line drawn, the code uses **cv.line** to create a line between two specified landmark points. The first argument is the input image, and the second and third arguments are the coordinates of the two endpoints of the line.
- **(0, 0, 0) represents** the color black, and **(255, 255, 255) represents** the color white. The lines are drawn with **different line widths** to create a visual effect.
- **6 and 2** are the thickness of line.



```
if index == 0:  # 手首1
    cv.circle(image, (landmark[0], landmark[1]), 5, (255, 255, 255),
              -1)
    cv.circle(image, (landmark[0], landmark[1]), 5, (0, 0, 0), 1)
```

- **image:** The input image on which the circle will be drawn.
- **(landmark[0], landmark[1]):** The center coordinates of the circle. These coordinates are extracted from the landmark list, which likely represents a specific point on the hand.
- **8:** The radius of the circle in pixels.
  (255, 255, 255): The color of the circle. In this case, (255, 255, 255) represents white color.
- **(0, 0, 0):** The color of the circle. In this case, (0, 0, 0) represents black color.
- **-1:** The thickness parameter. When the thickness is set to a negative value (like -1), it indicates that the circle should be filled with the specified color rather than drawing an outline.
- **1:** The thickness of the circle's outline.

Drawing Bounding Rectangle

```python
def draw_bounding_rect(use_brect, image, brect):
    if use_brect:
        # Outer rectangle
        cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[3]),
                     (0, 0, 0), 1)

    return image
```

- **use_brect:** A boolean flag indicating whether to draw the bounding rectangle or not.
- **image:** The input image on which the bounding rectangle will be drawn.
- **brect:** A list containing the coordinates of the bounding rectangle. It's expected to have four values: (x, y, x + width, y + height).
  Inside the function:
- The use_brect flag is checked. If it's True, the following code is executed:
  - **cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[3]), (0, 0, 0), 1):** This line of draws a black rectangle on the image. The **(brect[0], brect[1])** coordinates represent the top-left corner of the rectangle, and **(brect[2], brect[3])** represent the bottom-right corner. The **(0, 0, 0)** specifies the color of the rectangle (black), and **1** is the thickness of the rectangle's outline.

Drawing text (about hand:left or right, hand sign and finger gesture test)

```python
def draw_info_text(image, brect, handedness, hand_sign_text,
                   finger_gesture_text):
    cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[1] - 22),
                 (0, 0, 0), -1)

    info_text = handedness.classification[0].label[0:]
    if hand_sign_text != "":
        info_text = info_text + ':' + hand_sign_text
    cv.putText(image, info_text, (brect[0] + 5, brect[1] - 4),
               cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1, cv.LINE_AA)

    if finger_gesture_text != "":
        cv.putText(image, "Finger Gesture:" + finger_gesture_text, (10, 60),
                   cv.FONT_HERSHEY_SIMPLEX, 1.0, (0, 0, 0), 4, cv.LINE_AA)
        cv.putText(image, "Finger Gesture:" + finger_gesture_text, (10, 60),
                   cv.FONT_HERSHEY_SIMPLEX, 1.0, (255, 255, 255), 2,
                   cv.LINE_AA)

    return image
```

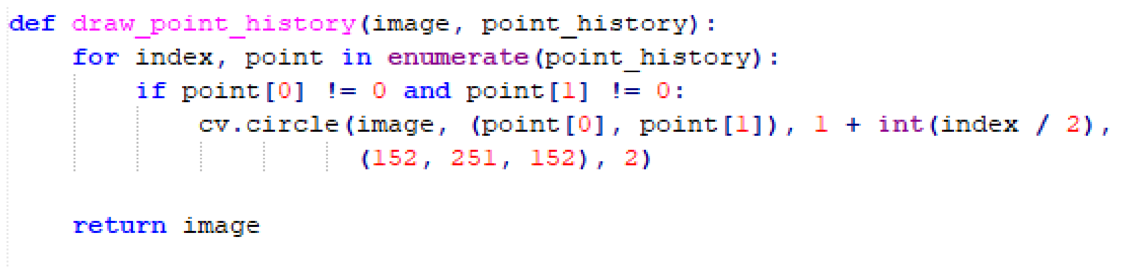
cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[1] - 22), (0, 0, 0), -1): This line creates a filled black rectangle above the bounding rectangle (specified by brect) in the image. The purpose of this rectangle is to serve as the background for the information text.
info_text = handedness.classification[0].label[0:]: This line extracts the label of the handedness classification from the handedness object. The [0] index indicates the first classification result (if there are multiple hands detected), and label[0:] retrieves the entire label string.
if hand_sign_text != "":: This conditional block checks if there is a hand sign classification text available (hand_sign_text). If it is not an empty string, the hand sign text is appended to the info_text string, separated by a colon.
cv.putText(image, info_text, (brect[0] + 5, brect[1] - 4), cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255,

255), 1, cv.LINE_AA): This line adds the info_text to the image as information text related to handedness and hand sign classification. The parameters for cv.putText are as follows:

- **image:** The image on which the text will be drawn.
- **info_text:** The text string containing the handedness and hand sign information.
- **(brect[0] + 5, brect[1] - 4):** The coordinates at which the text will be positioned.
- **cv.FONT_HERSHEY_SIMPLEX:** The font type.
- **0.6:** The font scale.
- **(255, 255, 255):** The text color (white).
- **1:** The font thickness.
- **cv.LINE_AA:** The line type

if finger_gesture_text != "":
If there is a finger gesture classification text available **(finger_gesture_text)**, it will be added to the image twice: once with a thicker black outline (for visibility) and once with a thinner white outline (to create a visual effect). The text is placed at coordinates **(10, 60)**.

Draw a series of circles on the image to visualize a historical sequence of points



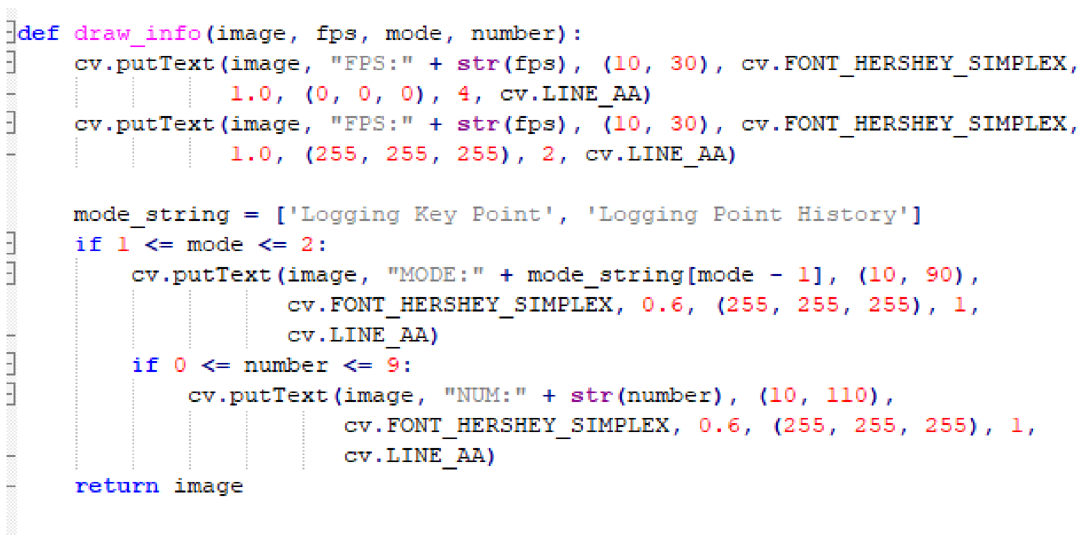
```python
def draw_point_history(image, point_history):
    for index, point in enumerate(point_history):
        if point[0] != 0 and point[1] != 0:
            cv.circle(image, (point[0], point[1]), 1 + int(index / 2),
                      (152, 251, 152), 2)

    return image
```

The function iterates over each point in the point_history list using a for loop and the enumerate function. The loop variable index holds the current index of the point, and point holds the coordinates of the point.
Inside the loop, the condition if point[0] != 0 and point[1] != 0: checks if the point is not to the origin (0, 0). This is done to ensure that only valid points are considered for drawing.
If the point is valid, the cv.circle function is used to draw a circle on the image. The of cv.circle are as follows:

- image: The image on which to draw the circle.
- **(point[0], point[1]):** The coordinates of the center of the circle (x, y).
- **1 + int(index / 2):** The radius of the circle, which increases with the index of the point divided by 2. This leads to a gradually increasing circle size over time.
- **(152, 251, 152):** The color of the circle in the format (B, G, R). Here, it's a shade of
- **2:** The thickness of the circle's outline.

After iterating through all the points, the function returns the modified image with the added circles.

Information text to the image (frames per second (FPS), the current mode, and a numeric value)



```python
def draw_info(image, fps, mode, number):
    cv.putText(image, "FPS:" + str(fps), (10, 30), cv.FONT_HERSHEY_SIMPLEX,
               1.0, (0, 0, 0), 4, cv.LINE_AA)
    cv.putText(image, "FPS:" + str(fps), (10, 30), cv.FONT_HERSHEY_SIMPLEX,
               1.0, (255, 255, 255), 2, cv.LINE_AA)

    mode_string = ['Logging Key Point', 'Logging Point History']
    if 1 <= mode <= 2:
        cv.putText(image, "MODE:" + mode_string[mode - 1], (10, 90),
                   cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1,
                   cv.LINE_AA)
        if 0 <= number <= 9:
            cv.putText(image, "NUM:" + str(number), (10, 110),
                       cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1,
                       cv.LINE_AA)
    return image
```

1. **Two cv.putText calls** are used to add FPS information to the image. One call adds a thicker black text with a shadow effect, while the other adds a slightly smaller white text to create a highlight effect. The FPS value is displayed at the coordinates (10, 30).
2. **mode_string** is a list containing two mode strings: 'Logging Key Point' and 'Logging Point History'.
3. The code then checks if the mode is between 1 and 2 (inclusive) using the condition **if 1 <= mode <= 2:**. If the condition is met, it means that the mode is valid.
4. Inside the condition, the function adds text to indicate the current mode using the cv.putText function. The mode string is fetched from the mode_string list based on the mode value, and it's displayed at the coordinates (10, 90).
5. Another check is performed to see if the number is between 0 and 9 (inclusive) using the condition **if 0 <= number <= 9**:. If the condition is met, it means that the numeric value is valid.
6. If the numeric value is valid, the function adds text to indicate the numeric value using the cv.putText function. The numeric value is displayed as **'NUM:'** followed by the value itself, it's shown at the coordinates (10, 110).
7. Finally, the modified image with the added information text is returned.

The code snippet if __name__ == '__main__': is a common Python programming construct used to ensure that the code inside the block is only executed when the Python script is run as the main program and not when it's imported as a module into another script.