Fingertip Trajectory Control Robotic Arm

Orin board users can directly open the terminal and input the tutorial commands to run directly. Jetson-Nano board users need to enter the docker container first, then input the tutorial commands in the docker to start the program.

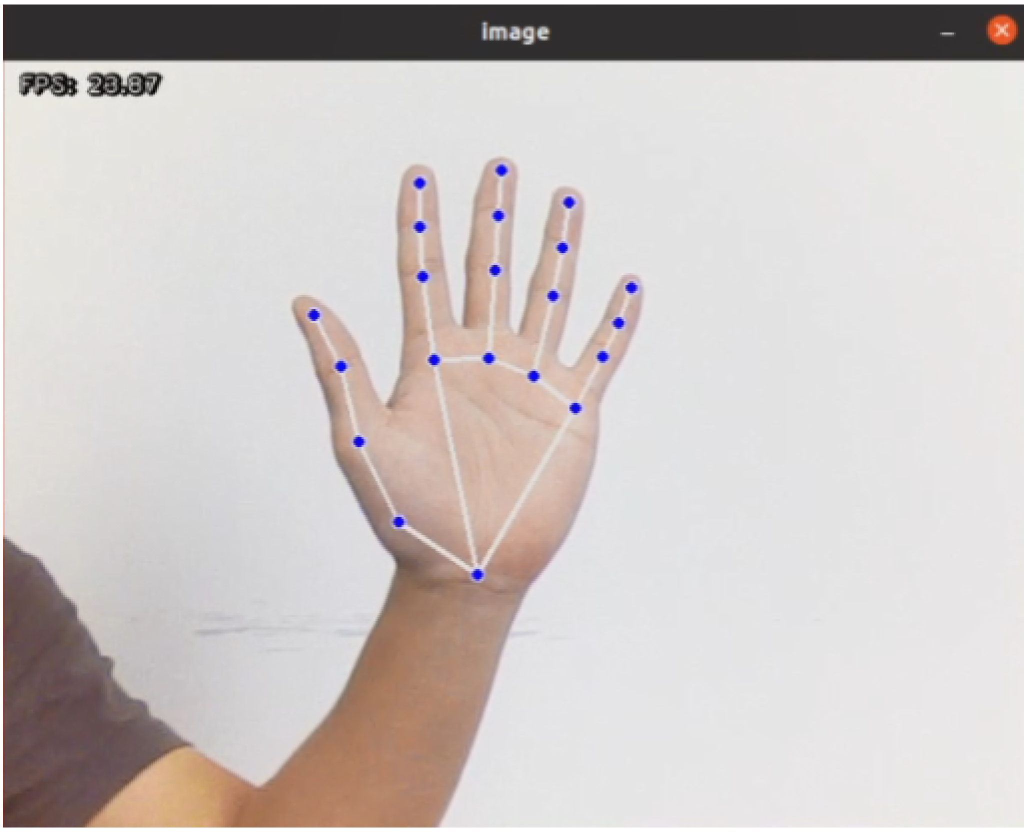
1. Introduction

The fingertip trajectory control robotic arm function is based on fingertip trajectory recognition, adding the capability of different trajectories to control different robotic arm actions.

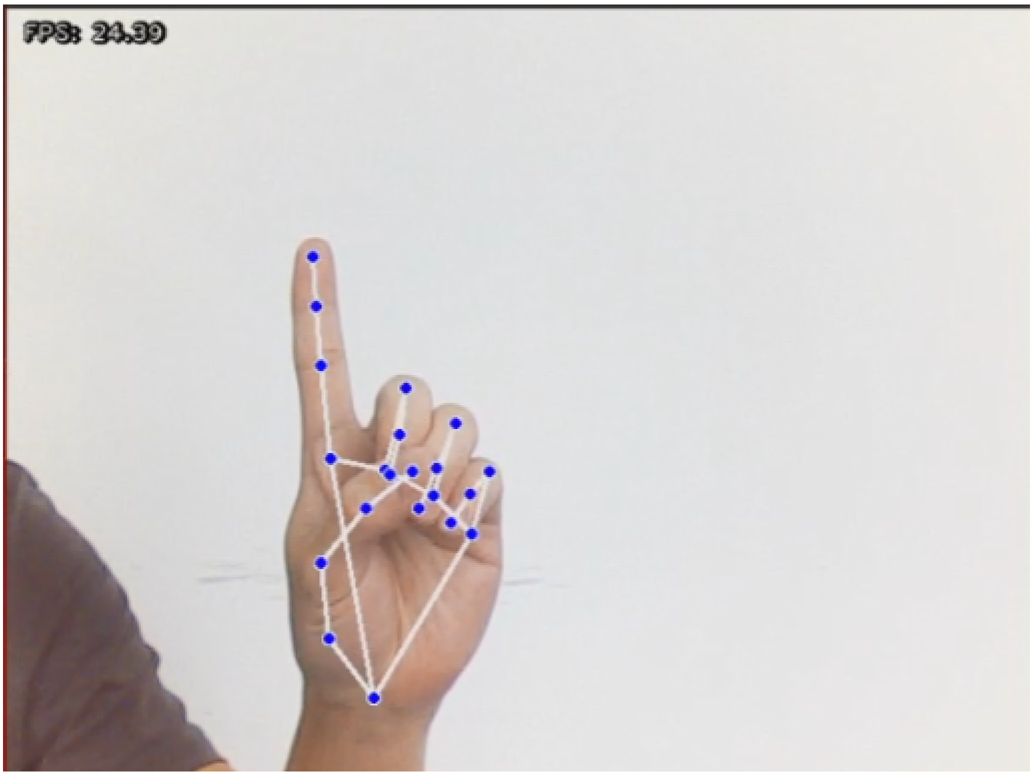
2. Launch

2.1. Program Description

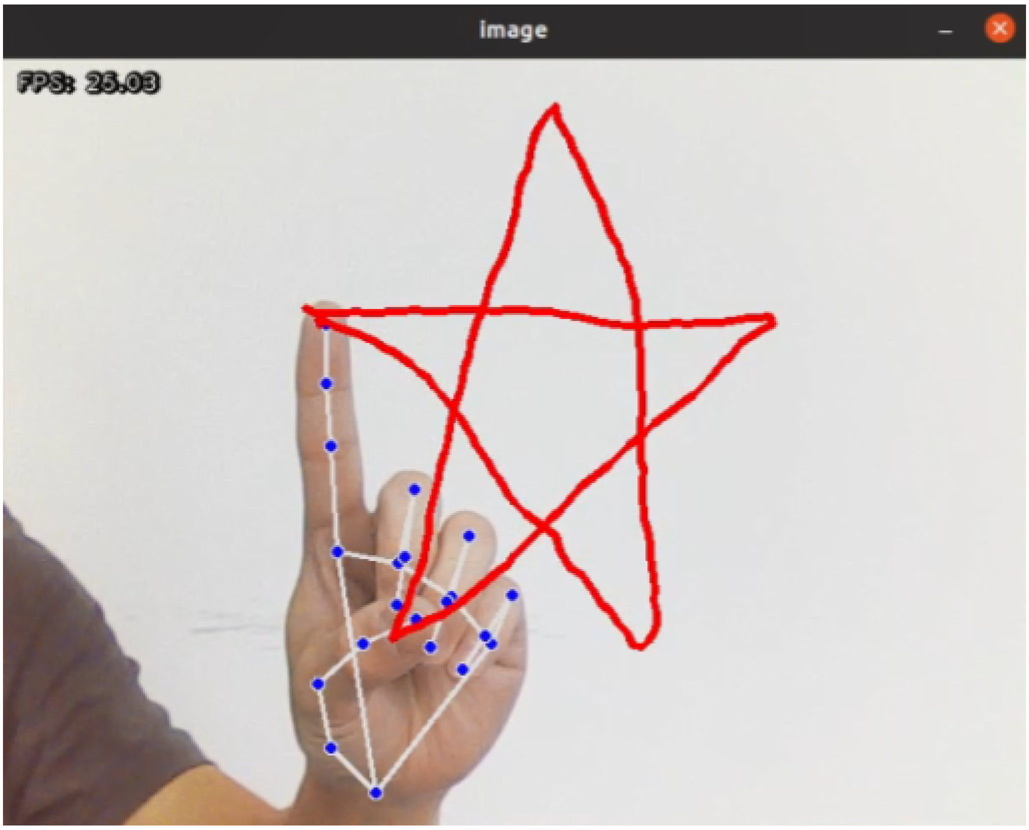
After the program starts, the camera captures images. Place your hand flat in the camera frame, with fingers spread open and palm facing the camera, similar to the number 5 gesture. The image will draw all the joints on the entire hand. Adjust the hand position to be in the upper middle part of the screen.



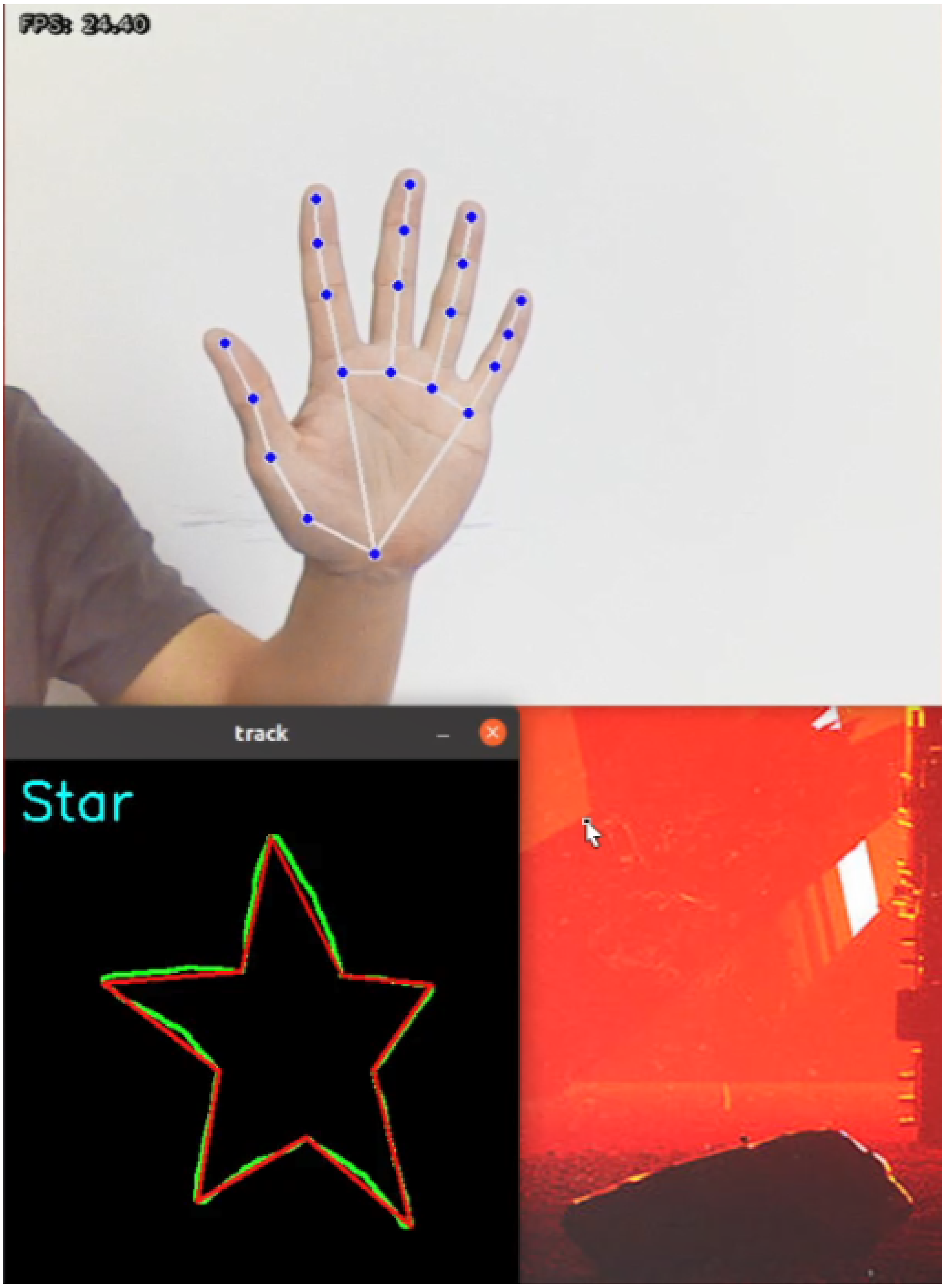
At this point, keep the index finger unchanged while retracting other fingers, similar to the number 1 gesture.



While maintaining the gesture 1 posture, move the finger position, and red lines will appear on the screen, drawing the path of the index finger movement.



When the drawing is complete, open all fingers, similar to the number 5 gesture, and the drawn shape will be generated at the bottom.



Note: The drawn shape needs to be in a closed state, otherwise some content may be missing.

Currently, there are four recognizable trajectory shapes: triangle, rectangle, circle, and five-pointed star.

When the camera recognizes different trajectory shapes, it will control the robotic arm to perform corresponding actions.

2.2. Program Launch

- Enter the following command to start the program

```
ros2 run dofbot_pro_mediapipe 16_FingerAction
```

Press the q key in the image or press Ctrl+c in the terminal to exit the program.

3. Source Code

Code path:

```
# Jetson-Nano users need to enter the docker container to view
~/dofbot_pro_ws/src/dofbot_pro_mediapipe/dofbot_pro_mediapipe/16_FingerAction.py
```

```
#!/usr/bin/env python3
# coding: utf8
import os
import enum
import cv2
import time
import numpy as np
import mediapipe as mp
import rclpy
from rclpy.node import Node
import queue
from sensor_msgs.msg import Image
from dofbot_utils.fps import FPS
import gc
from dofbot_utils.vutils import distance, vector_2d_angle, get_area_max_contour
import threading
from Arm_Lib import Arm_Device
from dofbot_utils.robot_controller import Robot_Controller
from cv_bridge import CvBridge  # Importing CvBridge


def get_hand_landmarks(img, landmarks):
    """
    Convert landmarks from MediaPipe's normalized output to pixel coordinates
    :param img: The image corresponding to pixel coordinates
    :param landmarks: Normalized key points
    :return:
    """
    h, w, _ = img.shape
    landmarks = [(lm.x * w, lm.y * h) for lm in landmarks]
    return np.array(landmarks)


def hand_angle(landmarks):
    """
    Calculate the bending angle of each finger
    :param landmarks: Hand key points
    :return: Angles of each finger
    """
    angle_list = []
    # thumb 大拇指
    angle_ = vector_2d_angle(landmarks[3] - landmarks[4], landmarks[0] -
landmarks[2])
    angle_list.append(angle_)
    # index 食指
    angle_ = vector_2d_angle(landmarks[0] - landmarks[6], landmarks[7] -
landmarks[8])
    angle_list.append(angle_)
    # middle 中指
    angle_ = vector_2d_angle(landmarks[0] - landmarks[10], landmarks[11] -
landmarks[12])
    angle_list.append(angle_)
    # ring 无名指
    angle_ = vector_2d_angle(landmarks[0] - landmarks[14], landmarks[15] -
landmarks[16])
    angle_list.append(angle_)
    # pink 小拇指
    angle_ = vector_2d_angle(landmarks[0] - landmarks[18], landmarks[19] -
landmarks[20])
    angle_list.append(angle_)
    angle_list = [abs(a) for a in angle_list]
    return angle_list


def h_gesture(angle_list):
    """
    Determine the gesture made by the finger through 2D features
    :param angle_list: Bending angles of each finger
    :return : Gesture name string
    """
    thr_angle, thr_angle_thumb, thr_angle_s = 65.0, 53.0, 49.0
    if (angle_list[0] < thr_angle_s) and (angle_list[1] < thr_angle_s) and
(angle_list[2] < thr_angle_s) and (
            angle_list[3] < thr_angle_s) and (angle_list[4] < thr_angle_s):
        gesture_str = "five"
    elif (angle_list[0] > 5) and (angle_list[1] < thr_angle_s) and
(angle_list[2] > thr_angle) and (
            angle_list[3] > thr_angle) and (angle_list[4] > thr_angle):
        gesture_str = "one"
    else:
        gesture_str = "none"
    return gesture_str


class State(enum.Enum):
    NULL = 0
    TRACKING = 1
    RUNNING = 2


def draw_points(img, points, tickness=4, color=(255, 0, 0)):
    """
    Draw the recorded connected points on the image
    """
    points = np.array(points).astype(dtype=np.int32)
    if len(points) > 2:
        for i, p in enumerate(points):
            if i + 1 >= len(points):
                break
            cv2.line(img, tuple(p), tuple(points[i + 1]), color, tickness)


def get_track_img(points):
    """
    Generate a trajectory image with black background and white lines using
recorded points
    """
    points = np.array(points).astype(dtype=np.int32)
    x_min, y_min = np.min(points, axis=0).tolist()
    x_max, y_max = np.max(points, axis=0).tolist()
    track_img = np.full([y_max - y_min + 100, x_max - x_min + 100, 1], 0,
dtype=np.uint8)
    points = points - [x_min, y_min]
    points = points + [50, 50]
    draw_points(track_img, points, 1, (255, 255, 255))
    return track_img


class FingerActionNode(Node):
    def __init__(self):
        super().__init__('finger_action')
        self.drawing = mp.solutions.drawing_utils
        self.timer = time.time()
        self.move_state = False

        self.hand_detector = mp.solutions.hands.Hands(
            static_image_mode=False,
            max_num_hands=1,
            min_tracking_confidence=0.05,
            min_detection_confidence=0.6
        )

        self.fps = FPS()  # FPS calculator
        self.state = State.NULL
        self.points = []
        self.start_count = 0
        self.no_finger_timestamp = time.time()

        self.gc_stamp = time.time()
        self.image_queue = queue.Queue(maxsize=1)
        self.bridge = CvBridge()

        # Initialize video capture device
        self.cap = cv.VideoCapture(0, cv.CAP_V4L2)
        self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
        self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
        if not self.cap.isOpened():
            self.get_logger().error("Error: Could not open video device.")
            rclpy.shutdown()

        self.Arm = Arm_Device()
        self.robot = Robot_Controller()
        self.robot.move_init_pose()

    def image_proc(self):
        ret, frame = self.cap.read()
        if not ret:
            self.get_logger().error("Error: Could not read frame from video
device.")
            return

        rgb_image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        rgb_image = cv2.flip(rgb_image, 1)  # Horizontal flip
        result_image = np.copy(rgb_image)
        result_call = None
        if self.timer <= time.time() and self.state == State.RUNNING:
            self.state = State.NULL
        try:
            results = self.hand_detector.process(rgb_image) if self.state !=
State.RUNNING else None
            if results is not None and results.multi_hand_landmarks:
                gesture = "none"
                index_finger_tip = [0, 0]
                self.no_finger_timestamp = time.time()  # Record current time
for timeout handling
                for hand_landmarks in results.multi_hand_landmarks:
                    self.drawing.draw_landmarks(
                        result_image,
                        hand_landmarks,
                        mp.solutions.hands.HAND_CONNECTIONS)
                    landmarks = get_hand_landmarks(rgb_image,
hand_landmarks.landmark)
                    angle_list = (hand_angle(landmarks))
                    gesture = (h_gesture(angle_list))
                    index_finger_tip = landmarks[8].tolist()

                if self.state == State.NULL:
                    if gesture == "one":  # Detect index finger extended alone,
other fingers clenched
                        self.start_count += 1
                        if self.start_count > 20:
                            self.state = State.TRACKING
                            self.points = []
                    else:
                        self.start_count = 0

                elif self.state == State.TRACKING:
                    if gesture == "five":  # Extend five fingers to end drawing
                        self.state = State.NULL

                        # Generate black and white trajectory image
                        track_img = get_track_img(self.points)
                        contours = cv2.findContours(track_img,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)[-2]
                        contour = get_area_max_contour(contours, 300)
                        contour = contour[0]
                        # Recognize the drawn shape according to trajectory
image
                        # cv2.fillPoly draws and fills polygons on the image
                        track_img = cv2.fillPoly(track_img, [contour, ], (255,
255, 255))
                        for _ in range(3):
                            # Erosion function
                            track_img = cv2.erode(track_img,
cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5)))
                            # Dilation function
                            track_img = cv2.dilate(track_img,
cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5)))
                        contours = cv2.findContours(track_img,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)[-2]
                        contour = get_area_max_contour(contours, 300)
                        contour = contour[0]
                        h, w = track_img.shape[:2]
                        track_img = np.full([h, w, 3], 0, dtype=np.uint8)
                        track_img = cv2.drawContours(track_img, [contour, ], -1,
(0, 255, 0), 2)
                        # Perform polygon fitting on image contour points
                        approx = cv2.approxPolyDP(contour, 0.026 *
cv2.arcLength(contour, True), True)
                        track_img = cv2.drawContours(track_img, [approx, ], -1,
(0, 0, 255), 2)
                        graph_name = 'unknown'
                        print(len(approx))
                        # Determine the shape based on the number of vertices of
the contour envelope
                        if len(approx) == 3:
                            graph_name = 'Triangle'
                        if len(approx) == 4 or len(approx) == 5:
                            graph_name = 'Square'
                        if 5 < len(approx) < 10:
                            graph_name = 'Circle'
                        if len(approx) == 10:
                            graph_name = 'Star'
                        cv2.putText(track_img, graph_name, (10, 40),
cv2.FONT_HERSHEY_SIMPLEX, 1.2, (255, 255, 0), 2)
                        cv2.imshow('track', track_img)
                        if not self.move_state:
                            self.move_state = True
                            task = threading.Thread(target=self.arm_move_action,
name="arm_move_action", args=(graph_name,))
                            task.setDaemon(True)
                            task.start()

                    else:
                        if len(self.points) > 0:
                            if distance(self.points[-1], index_finger_tip) > 5:
                                self.points.append(index_finger_tip)
                        else:
                            self.points.append(index_finger_tip)

                    draw_points(result_image, self.points)
                else:
                    pass
            else:
                if self.state == State.TRACKING:
                    if time.time() - self.no_finger_timestamp > 2:
                        self.state = State.NULL
                        self.points = []

        except BaseException as e:
            self.get_logger().error("e = {}".format(e))

        self.fps.update_fps()
        self.fps.show_fps(result_image)
        result_image = cv2.cvtColor(result_image, cv2.COLOR_RGB2BGR)
        cv2.imshow('image', result_image)
        key = cv2.waitKey(1)

        if key == ord(' '):  # Press space to clear recorded trajectory
            self.points = []
```

```python
        if time.time() > self.gc_stamp:
            self.gc_stamp = time.time() + 1
            gc.collect()

    def arm_move_triangle(self):
        self.Arm.Arm_serial_servo_write6_array([90, 131, 52, 0, 90, 180], 1500)
        time.sleep(1.5)
        self.Arm.Arm_serial_servo_write6_array([45, 180, 0, 0, 90, 180], 1500)
        time.sleep(2)
        self.Arm.Arm_serial_servo_write6_array([135, 180, 0, 0, 90, 180], 1500)
        time.sleep(2)
        self.Arm.Arm_serial_servo_write6_array([90, 131, 52, 0, 90, 180], 1500)
        time.sleep(1.5)

    def arm_move_square(self):
        self.Arm.Arm_serial_servo_write6_array(self.robot.P_ACTION_4, 1500)
        time.sleep(1.4)
        for i in range(3):
            self.Arm.Arm_serial_servo_write(4, -15, 300)
            time.sleep(0.4)
            self.Arm.Arm_serial_servo_write(4, 20, 300)
            time.sleep(0.4)

    def arm_move_circle(self):
        for i in range(5):
            self.Arm.Arm_serial_servo_write(5, 60, 300)
            time.sleep(0.4)
            self.Arm.Arm_serial_servo_write(5, 120, 300)
            time.sleep(0.4)
        self.Arm.Arm_serial_servo_write(5, 90, 300)
        time.sleep(0.4)

    def arm_move_star(self):
        for i in range(3):
            self.Arm.Arm_serial_servo_write6_array(self.robot.P_ACTION_3, 1200)
            time.sleep(1.2)
            self.Arm.Arm_serial_servo_write6_array(self.robot.P_LOOK_AT, 1000)
            time.sleep(1)

    def arm_move_action(self, name):
        self.Arm.Arm_Buzzer_On(1)
        time.sleep(1)
        if name == 'Triangle':
            self.arm_move_triangle()
        elif name == 'Square':
            self.arm_move_square()
        elif name == 'Circle':
            self.arm_move_circle()
        elif name == 'Star':
            self.arm_move_star()
        self.robot.move_init_pose()
        time.sleep(1.5)
        self.move_state = False


def main(args=None):
    rclpy.init(args=args)
    finger_node = FingerActionNode()
    try:
        while rclpy.ok():
            finger_node.image_proc()
    except KeyboardInterrupt:
        pass
    finally:
        finger_node.cap.release()
        cv2.destroyAllWindows()
        rclpy.shutdown()


if __name__ == "__main__":
    main()
```