ECEN524 Project 2

Written by: Aly Khater and Wesley Tu

Code utilizes the use of ChatGPT. Thanks ChatGPT.
Google Colab Notebook:
https://colab.research.google.com/drive/1-Pcez6uoeOhVOtL8-Ahw5ZQBjVQFKVH3?usp=sharing

Part A:

Scenario: Robot takes 4 initial objects: fork, knife, spoon, and bowl, and places them into their correct positions on the table.

4 objects:

1) Fork
2) Knife
3) Spoon
4) Bowl

Link to Google Drive that contains video::
https://drive.google.com/drive/folders/1sljwpXch5amUFCoxIKjRxgJ9iSvdbtCW?usp=sharing
Have to upload the files directly to the content(default) directory in Google Colab.

Took inspiration from basic dining table settings. The robot is setting up the table for potential diners to come in and enjoy a hearty, exquisite, high-end, and extremely expensive meal for Dr. Maria Kyrarini.

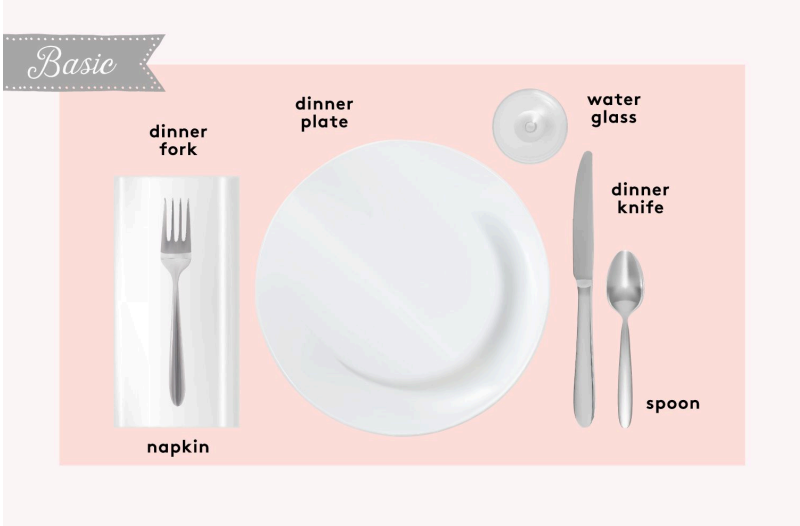

Figure 1: Basic Dining Table Setting

Part B:

1) Identify a pretrained model or develop your own method that can detect the objects in your video. Provide details of the model you used or the method you developed and screenshots of the code and the results. [20%]
Took a YOLO11 image recognition model which was pre-trained on the COCO dataset, which can recognize many different objects, including forks, knives, spoons, and bowls. We took a sample of 1 frame per second of the video, to get much faster times and results.

Code:

```
import cv2
import matplotlib.pyplot as plt
from ultralytics import YOLO

def process_video(video_path, model, frame_interval=1000):
    """
    Processes the video file, analyzing one frame per
`frame_interval` milliseconds.

    :param video_path: Path to the video file
    :param model: YOLO model instance
    :param frame_interval: Time interval in milliseconds between
frames (default: 1000ms = 1 second)
    """
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        print(f"Error opening video file: {video_path}")
        return

    fps = cap.get(cv2.CAP_PROP_FPS)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    duration = total_frames / fps if fps > 0 else 0  # Calculate
total video duration in seconds

    print(f"Video FPS: {fps}, Total Frames: {total_frames},
Duration: {duration:.2f} seconds")

    current_time = 0  # Start at 0ms
```

```
    while current_time < duration * 1000:  # Convert duration to
milliseconds
        cap.set(cv2.CAP_PROP_POS_MSEC, current_time)  # Move to the
correct time
        ret, frame = cap.read()
        if not ret:
            break

        results = model(frame)
        result = results[0]

        for *xyxy, conf, cls in result.boxes.data.tolist():
            x1, y1, x2, y2 = int(xyxy[0]), int(xyxy[1]),
int(xyxy[2]), int(xyxy[3])
            label = f'{result.names[int(cls)]} {conf:.2f}'
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
            cv2.putText(frame, label, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)

        # Display the frame
        plt.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
        plt.axis('off')
        plt.show()

        current_time += frame_interval  # Move to the next timestamp

    cap.release()
    cv2.destroyAllWindows()

# Example usage
model = YOLO('yolo11n.pt') # Pretrained on COCO dataset
video_path = "/content/setting.mp4"  # Replace with your video file
path
process_video(video_path, model, frame_interval=1000)  # Process one
frame per second
```

Video images:

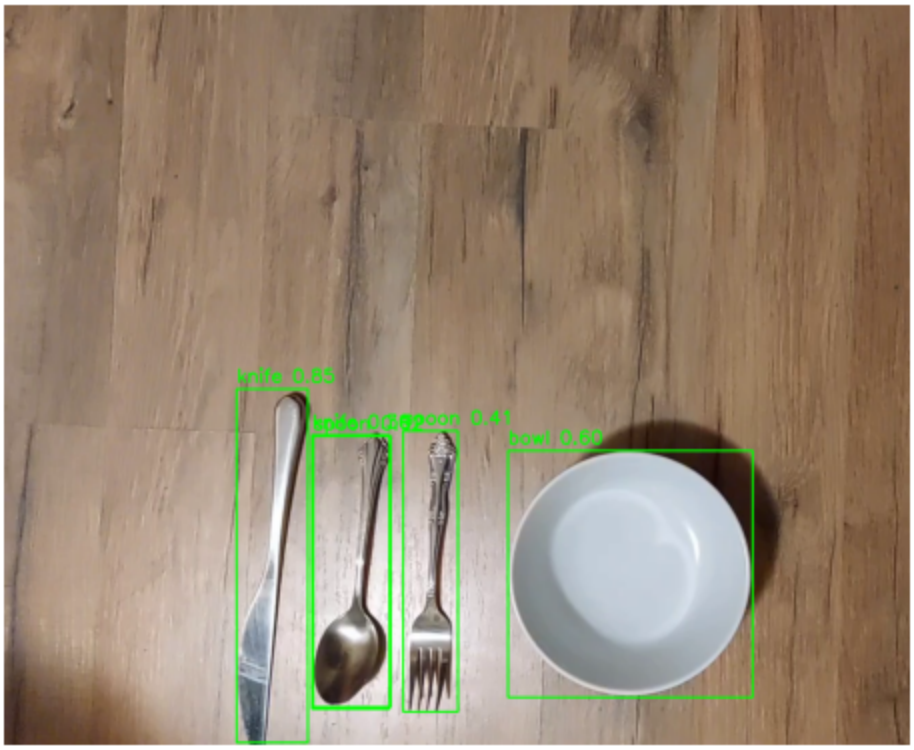

Figure 2: Initial Frame of the video. Objects are at their initial positions.

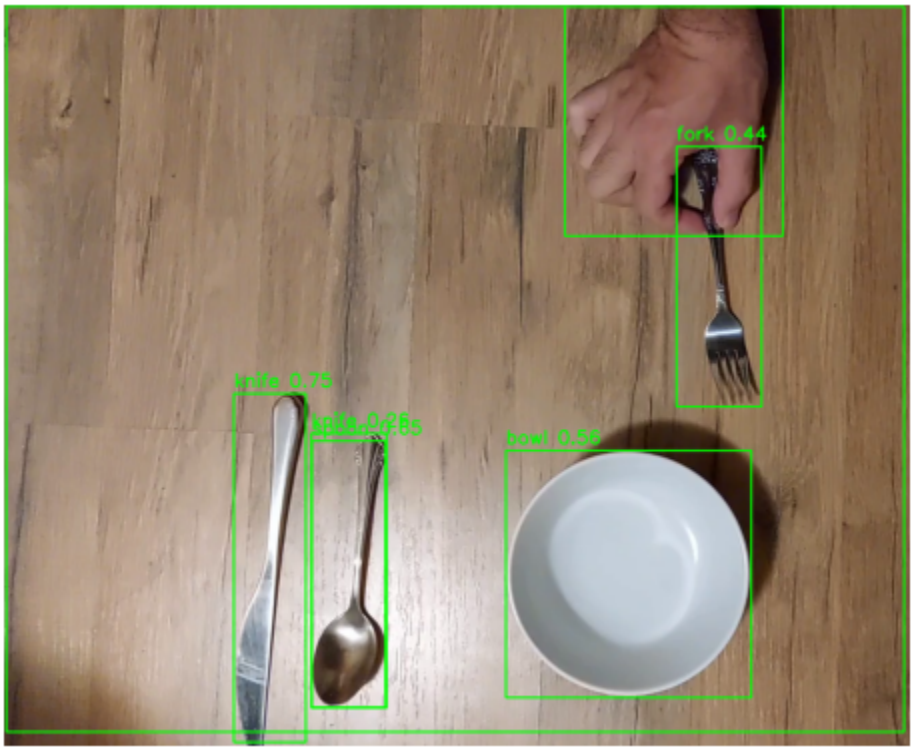

Figure 3: Fork moved to final position

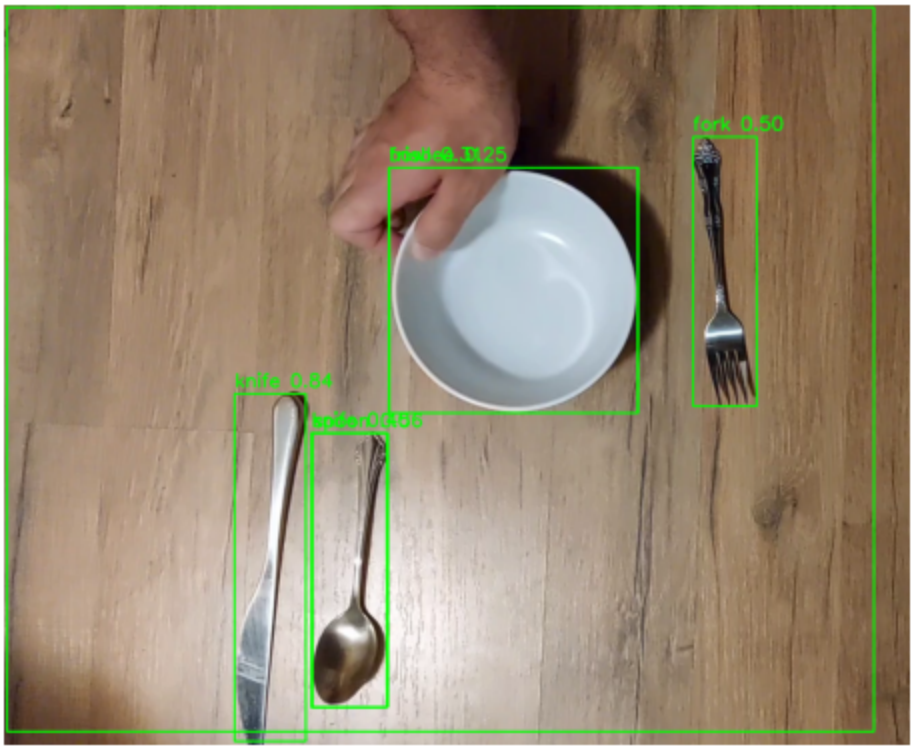

Figure 4: Bowl moved to final position

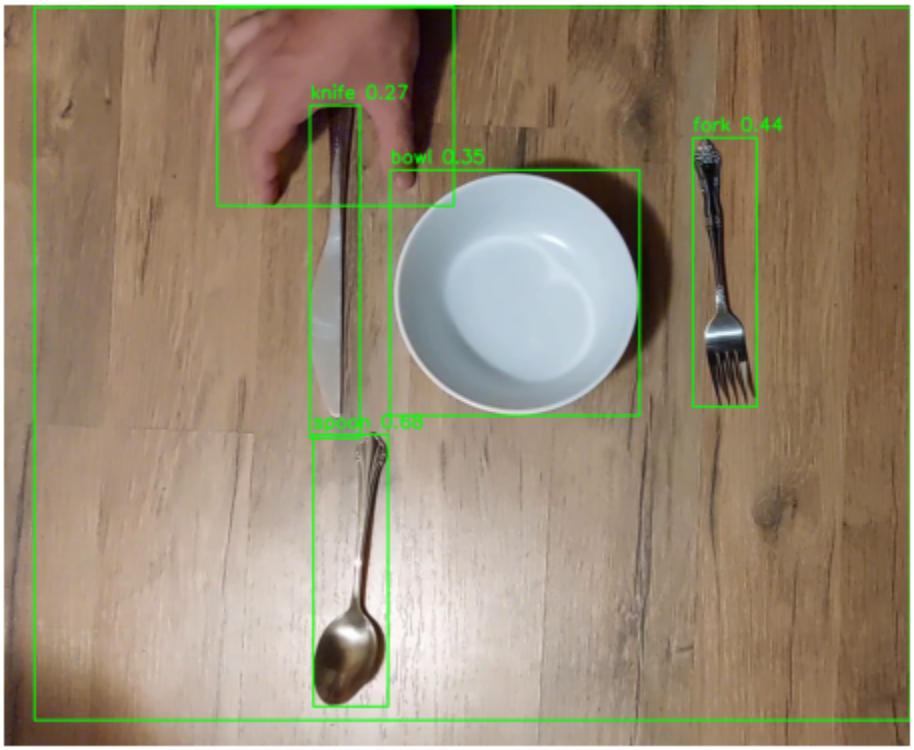

Figure 5: Knife moved to final position

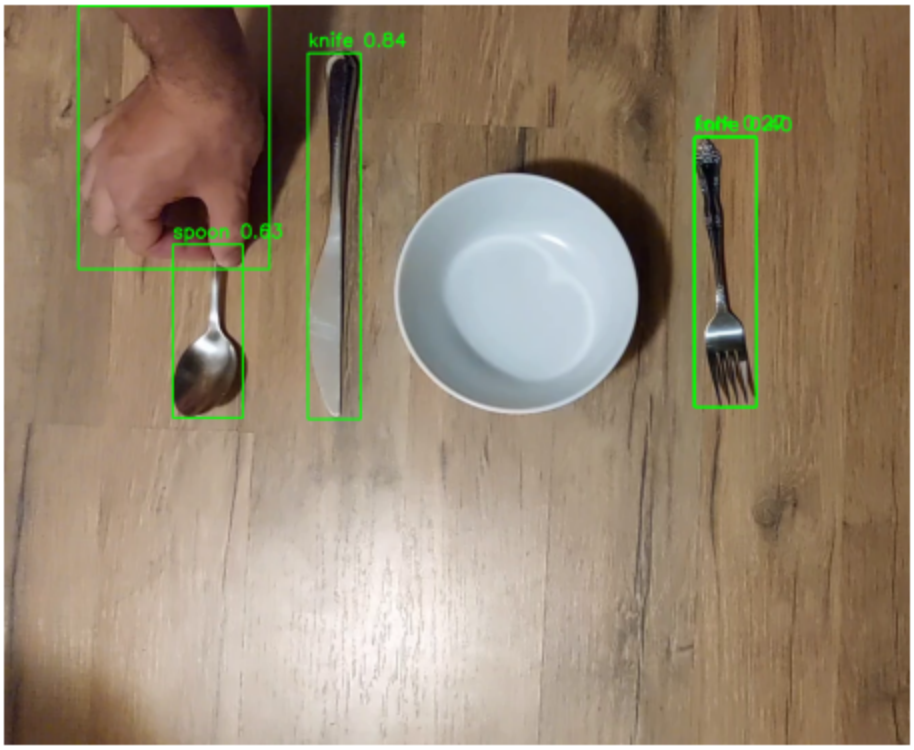

Figure 6: Spoon moved to final position

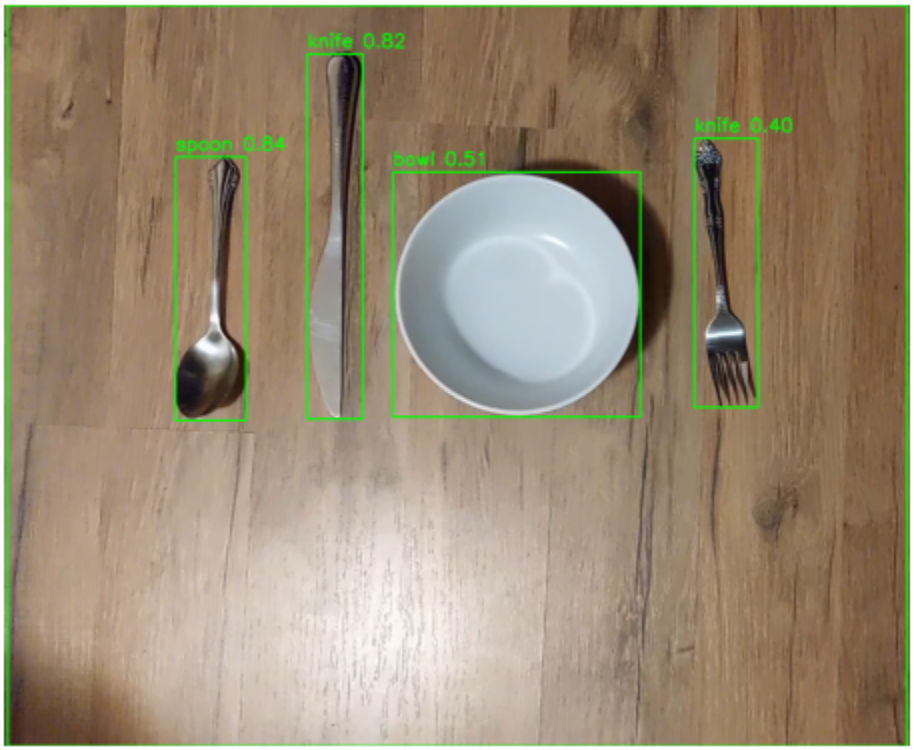

Figure 7: Final frame of the video

2) Identify a pretrained model or develop your own method that can recognize the actions you performed in the video (e.g. grasping, moving, releasing, etc.). Provide details of the model you used or the method you developed and screenshots of the code and the results. [20%]
Used MediaPipe, which is a selection of libraries used in many applications related to artificial intelligence and machine learning. We specifically used MediaPipe for its hand landmarks detection, since we could use the hand landmarks to detect whether or not we were holding an object.

Code:

```
import cv2
import mediapipe as mp
import matplotlib.pyplot as plt
from ultralytics import YOLO
import numpy as np

# Initialize MediaPipe Hands
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(static_image_mode=False, max_num_hands=2,
min_detection_confidence=0.5)
mp_drawing = mp.solutions.drawing_utils

def detect_gesture(hand_landmarks):
    """
    Detects if the hand is 'open' or 'pinching' using thumb and
index finger distance.
    """
    if hand_landmarks:
        # Thumb tip (landmark 4) & Index fingertip (landmark 8)
        thumb_tip = np.array([hand_landmarks.landmark[4].x,
hand_landmarks.landmark[4].y])
        index_tip = np.array([hand_landmarks.landmark[8].x,
hand_landmarks.landmark[8].y])

        # Calculate Euclidean distance
        distance = np.linalg.norm(thumb_tip - index_tip)

        # Pinch threshold (adjust as needed)
        return "Pinching" if distance < 0.05 else "Open"

    return "Unknown"
```

```
def process_video(video_path, model, frame_interval=1000):
    """
    Processes the video, analyzing one frame per `frame_interval`
milliseconds.

    :param video_path: Path to the video file
    :param model: YOLO model instance
    :param frame_interval: Time interval in milliseconds between
frames (default: 1000ms = 1 second)
    """
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        print(f"Error opening video file: {video_path}")
        return

    fps = cap.get(cv2.CAP_PROP_FPS)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    duration = total_frames / fps if fps > 0 else 0  # Convert
frames to seconds

    print(f"Video FPS: {fps}, Total Frames: {total_frames},
Duration: {duration:.2f} sec")

    current_time = 0  # Start at 0ms

    while current_time < duration * 1000:  # Convert duration to
milliseconds
        cap.set(cv2.CAP_PROP_POS_MSEC, current_time)  # Move to next
frame at correct time
        ret, frame = cap.read()
        if not ret:
            break

        # Convert BGR to RGB for MediaPipe
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        result = hands.process(rgb_frame)

        # Detect hand gesture
        gesture = "Unknown"
```

```
        if result.multi_hand_landmarks:
            gesture = detect_gesture(result.multi_hand_landmarks[0])

            # Draw hand landmarks
            for hand_landmarks in result.multi_hand_landmarks:
                mp_drawing.draw_landmarks(frame, hand_landmarks,
mp_hands.HAND_CONNECTIONS)

        # YOLO object detection
        results = model(frame)
        yolo_result = results[0]

        for *xyxy, conf, cls in yolo_result.boxes.data.tolist():
            x1, y1, x2, y2 = int(xyxy[0]), int(xyxy[1]),
int(xyxy[2]), int(xyxy[3])
            label = f'{yolo_result.names[int(cls)]} {conf:.2f}'
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
            cv2.putText(frame, label, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)

        # Display Gesture Recognition
        cv2.putText(frame, f"Gesture: {gesture}", (30, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)

        # Show frame (processes one frame per `frame_interval`)
        plt.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
        plt.axis('off')
        plt.show()

        current_time += frame_interval  # Move to the next second

    cap.release()
    cv2.destroyAllWindows()

# Load YOLO Model
model = YOLO('yolo11n.pt')

# Run video processing
video_path = "/content/setting.mp4"
```

```
process_video(video_path, model, frame_interval=1000)  # 1 frame per
second
```

Video images:

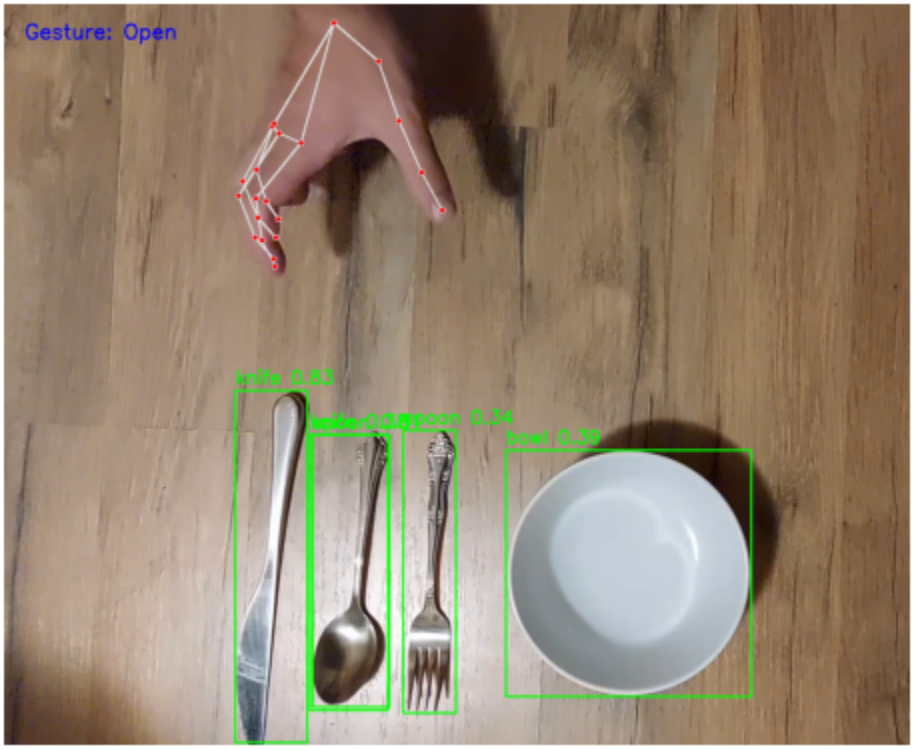

Figure 8: Hand detected as being open

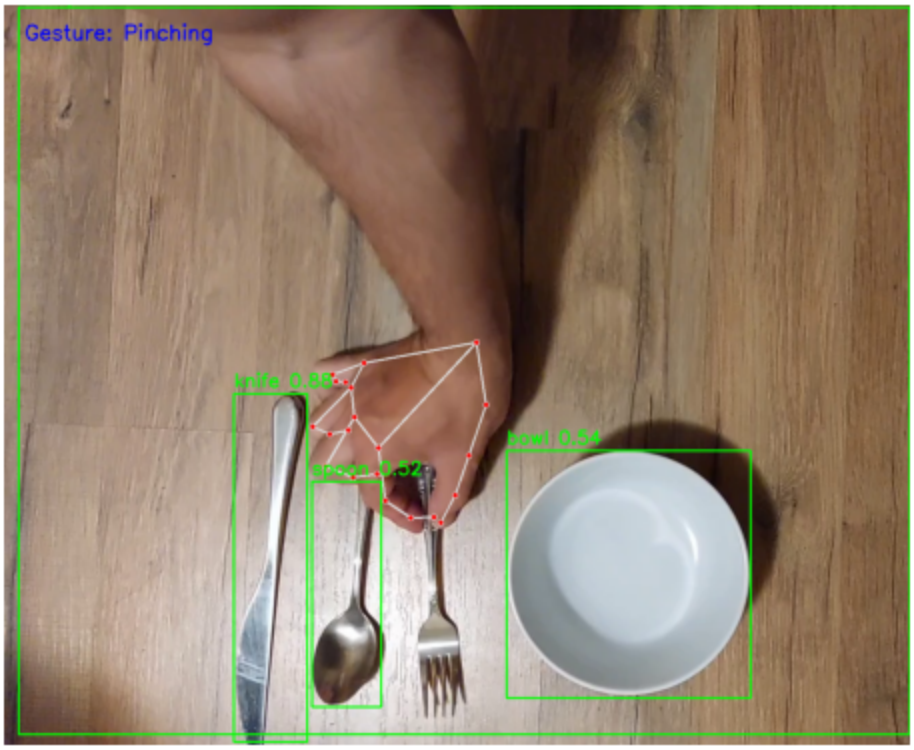

Figure 9: Hand detected as pinching (grasping) the fork

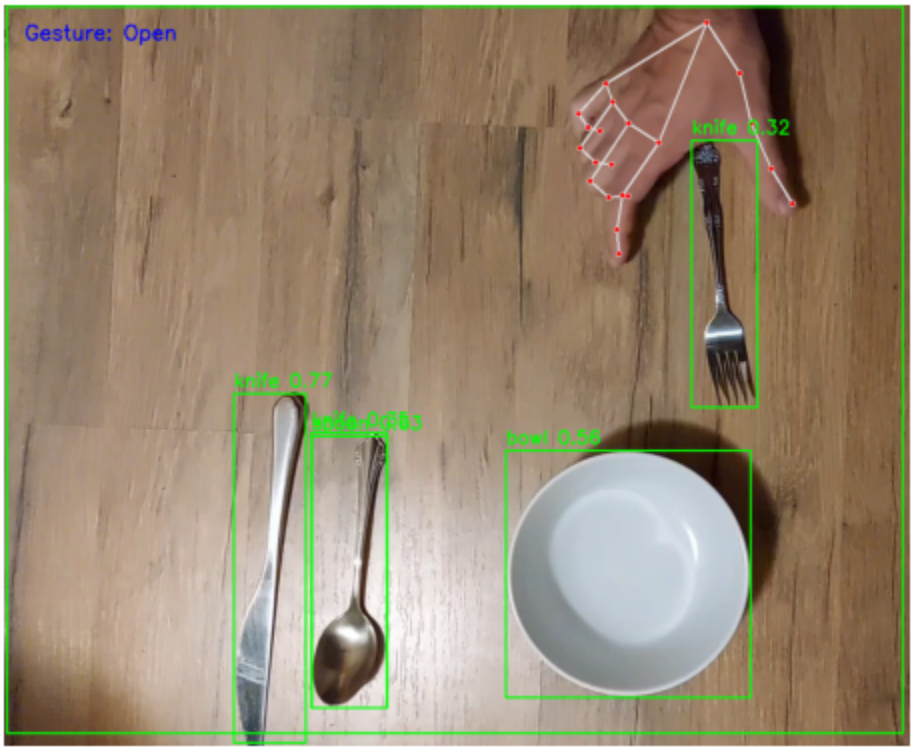

Figure 10: Hand open, fork moved to final position

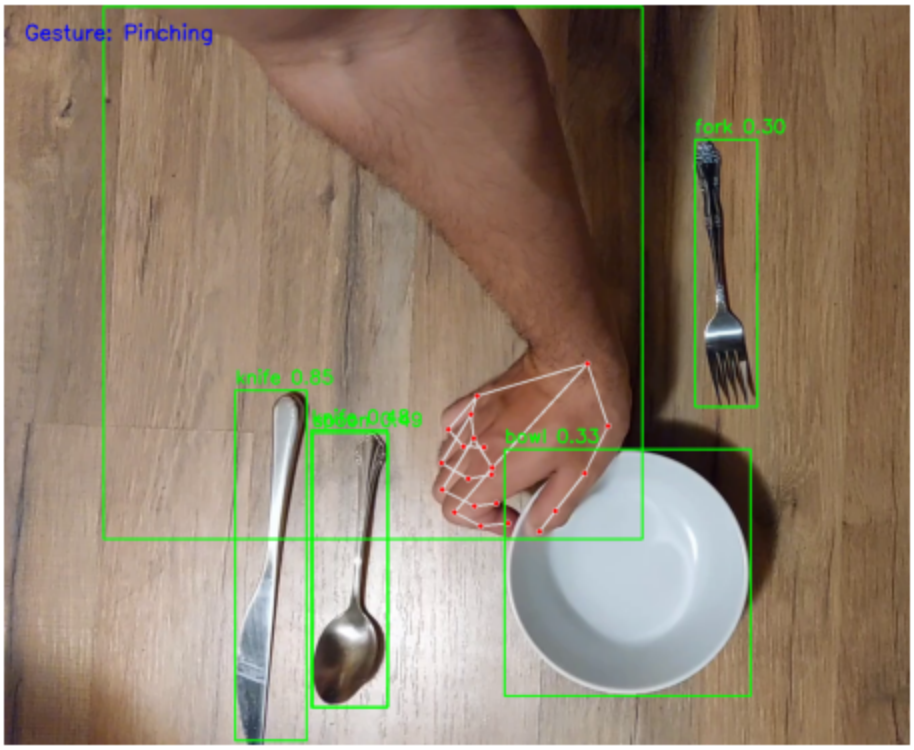

Figure 11: Hand pinching. At bowl initial position.

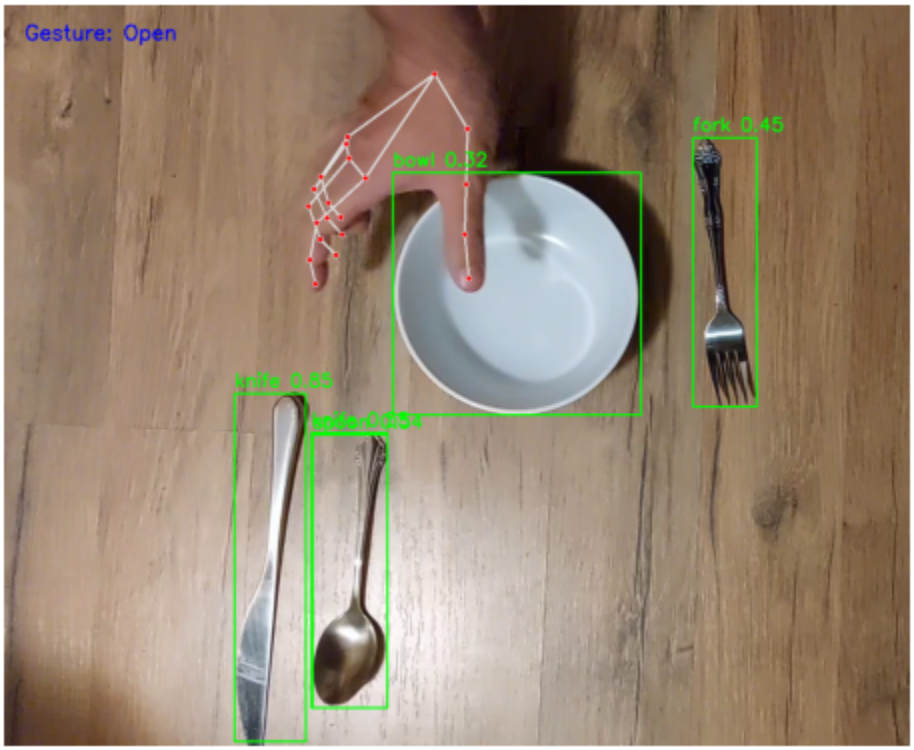

Figure 12: Hand open, bowl moved to final position

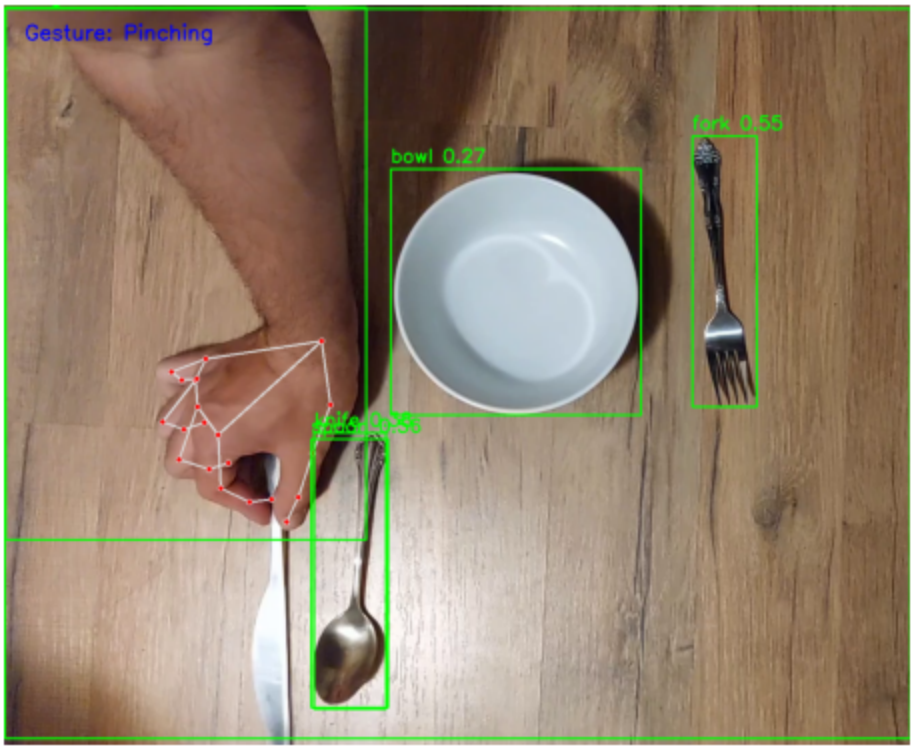

Figure 13: Hand pinching. At knife initial position

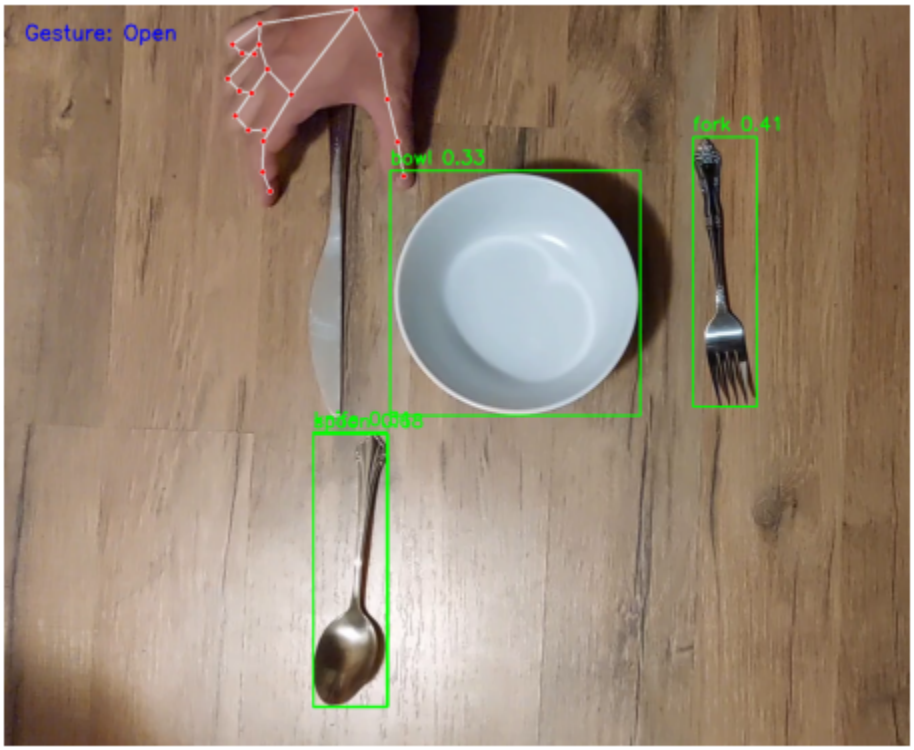

Figure 14: Hand open. At knife final position

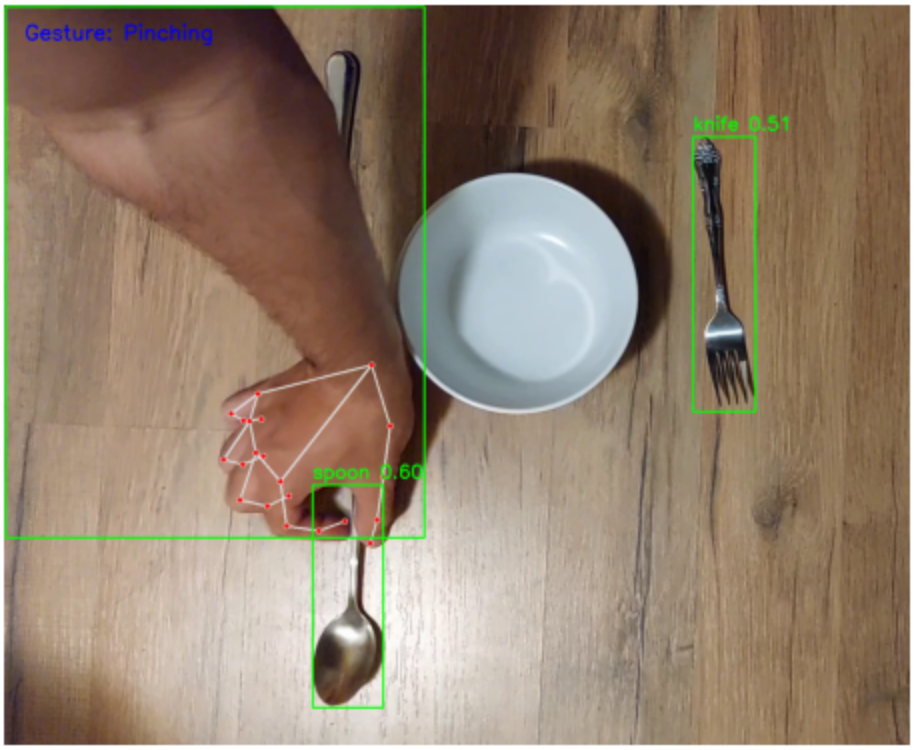

Figure 15: Hand pinching. At spoon initial position

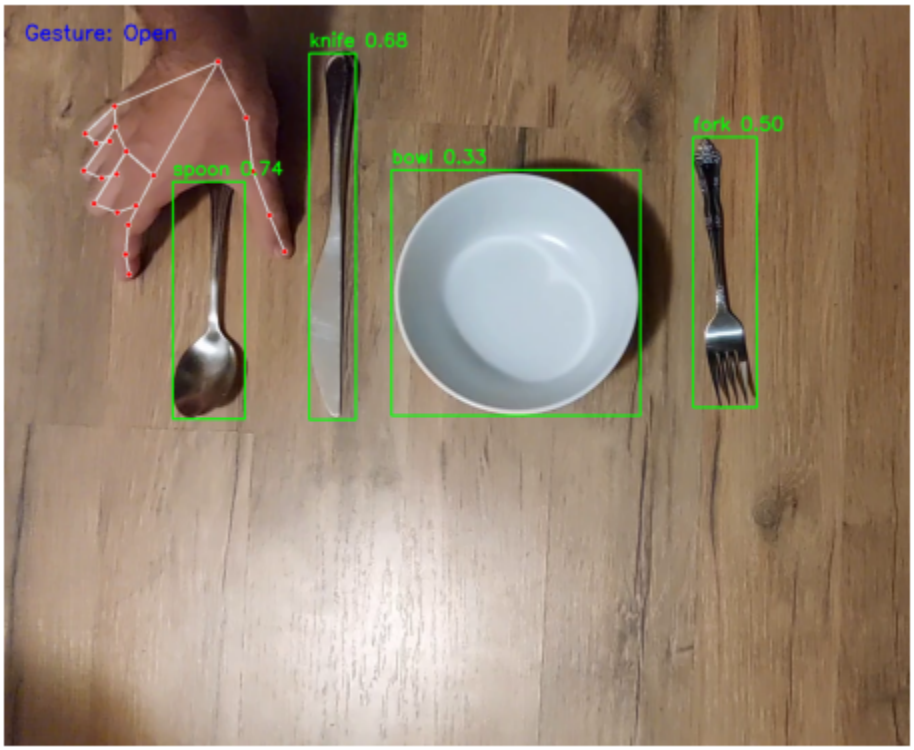

Figure 16: Hand open. At spoon final position

Part C:
Identify a method or develop your own method that can generate a sequence of actions when applied to your videos. Explain the method you used/developed and provide screenshots of the code and the results. Discuss the limitations of your approach if extended to other scenarios.
We saved a csv file of the object positions by utilizing the bounding boxes of each object and saving the center of these bounding boxes to determine the position of the objects. The csv file contains the x and y positions of each object, and also determines whether the hand is open, grasping, or unknown. We defaulted the "unknown" value as "Open" when reading the csv file.
In reading this csv file, we are able to identify the key frames in which an object is being manipulated. An example of this is shown in Figure 17.

Frame	x-bowl	x-fork	x-spoon	x-knife	y-bowl	y-fork	y-spoon	y-knife	Hand Gesture
0	913.5		622.0	505.5	830.0		826.0	827.0	Unknown
1	912.5		622.0	504.5	830.0		826.0	827.0	Unknown
2	911.0		620.5	503.0	829.5		827.0	828.0	Open
3	910.5	616.0	621.5	618.0	829.5	823.0	826.5	824.0	Open
4	910.5	615.5	502.0	388.0	830.0	821.0	863.5	822.0	Open
5	911.0		499.5	388.5	830.5		859.5	821.0	Pinching
6	910.0	634.5	501.0	388.5	830.0	884.0	848.0	821.5	Pinching
7	911.0		504.5	503.5	829.5		824.5	824.0	Pinching
8	910.5		503.5	503.5	829.0		828.0	823.0	Pinching
9	910.5	1063.0	503.5	502.5	829.0	454.0	828.0	824.0	Pinching
10	911.0		504.0	503.5	829.0		828.0	823.5	Pinching
11	910.5		504.0	1051.0	829.0		826.0	391.5	Open

Figure 17: First 11 seconds of the csv file.

The empty cells of the csv file are due to misclassification of the objects in the video, but as long as there exists a value for the position in the video in that pinching phase, we were able to identify how the object moves.
As shown above, we have a "Pinching" phase between frames 5-10. The object that moved the most in that phase is the fork, so we determined that the fork was the object that was being moved in those frames. Figure X shows which object moved the most, and noting down its initial position and final position.

```
Tracking objects between frames 5 → 10
→ Object that moved the most: fork (607.05 px)

Tracking objects between frames 15 → 18
→ Object that moved the most: bowl (430.51 px)

Tracking objects between frames 23 → 24
→ Object that moved the most: knife (697.56 px)

Tracking objects between frames 31 → 34
→ Object that moved the most: spoon (352.57 px)

Filtered Object Movement Trajectories:
Fork moved from (634.5, 884.0) at Frame 5 to (1063.0, 454.0) at Frame 10
Bowl moved from (909.5, 828.0) at Frame 15 to (743.5, 434.0) at Frame 18
Knife moved from (504.5, 825.5) at Frame 23 to (1051.0, 392.0) at Frame 24
Spoon moved from (501.5, 862.5) at Frame 31 to (247.0, 618.5) at Frame 34
```

Figure 18: Tracking of objects that moved the most in Pinching phases. Tracks down the initial and final position of the objects in that pinching phase.

We have four pinching phases that can be shown above between the four objects. The code for tracking these object positions is shown below

```python
import pandas as pd
import numpy as np

# Load CSV file
csv_file = "object_positions.csv"
df = pd.read_csv(csv_file)

# Treat "Unknown" as "Open" for missing detections
df["Hand Gesture"] = df["Hand Gesture"].replace("Unknown", "Open")

# Object names to track
objects = ["knife", "spoon", "fork", "bowl"]

# Initialize variables for tracking pinching events
pinching_start = None
pinching_end = None
pinching_intervals = []

# Detect when pinching starts and ends
for i in range(1, len(df)):
    prev_gesture = df.loc[i - 1, "Hand Gesture"]
```

```
    curr_gesture = df.loc[i, "Hand Gesture"]

    if prev_gesture == "Open" and curr_gesture == "Pinching":
        pinching_start = df.loc[i, "Frame"]

    if prev_gesture == "Pinching" and curr_gesture == "Open":
        pinching_end = df.loc[i - 1, "Frame"]
        pinching_intervals.append((pinching_start, pinching_end))
        pinching_start, pinching_end = None, None  # Reset for next
sequence

# Analyze each pinching interval
filtered_trajectories = {}

for start, end in pinching_intervals:
    print(f"\nTracking objects between frames {start} → {end}")

    # Store positions frame by frame during pinching
    movement_distances = {obj: 0 for obj in objects}
    start_positions = {}
    end_positions = {}
    previous_positions = {obj: None for obj in objects}

    # Iterate through all frames in the pinching period
    for frame_num in range(start, end + 1):
        frame_data = df[df["Frame"] == frame_num]

        if not frame_data.empty:
            for obj in objects:
                x, y = frame_data[f"x-{obj}"].values[0],
frame_data[f"y-{obj}"].values[0]

                if not np.isnan(x) and not np.isnan(y):
                    # Store start position
                    if obj not in start_positions:
                        start_positions[obj] = (x, y)

                    # Compute movement distance from the previous frame
                    if previous_positions[obj] is not None:
                        prev_x, prev_y = previous_positions[obj]
```

```
                        movement_distances[obj] += np.sqrt((x - prev_x) **
2 + (y - prev_y) ** 2)

                    previous_positions[obj] = (x, y)
                    end_positions[obj] = (x, y)  # Update end position

    # Find the object that moved the most
    most_moved_object = max(movement_distances,
key=movement_distances.get)
    print(f"→ Object that moved the most: {most_moved_object}
({movement_distances[most_moved_object]:.2f} px)")

    # Store only start and end positions of the most moved object
    if most_moved_object in start_positions and most_moved_object in
end_positions:
        filtered_trajectories[most_moved_object] = (start, end,
start_positions[most_moved_object], end_positions[most_moved_object])

# Print movement trajectories (only start and end for the most moved
object)
print("\nFiltered Object Movement Trajectories:")
for obj, (start_frame, end_frame, start_pos, end_pos) in
filtered_trajectories.items():
    print(f"{obj.capitalize()} moved from {start_pos} at Frame
{start_frame} to {end_pos} at Frame {end_frame}")
```

With the object positions saved into a variable called filtered_trajectories, we are now able to generate a sequence of actions for the robot to move. We have three actions the robot can move:

- Move
- Pick
- Place

Here is the code to generate the actions for the robot.

```
# Robot starts at (0, 0)
robot_position = (0, 0)
robot_actions = []

# Generate movement sequence for the robot using filtered_trajectories
```

```
for obj, (start_frame, end_frame, initial_pos, final_pos) in
filtered_trajectories.items():
    # Move to object initial position
    robot_actions.append(f"Move from {robot_position} to {initial_pos}")

    # Pick up object
    robot_actions.append(f"Pick {obj} at {initial_pos}")

    # Move to object final position
    robot_actions.append(f"Move {obj} from {initial_pos} to {final_pos}")

    # Place object
    robot_actions.append(f"Place {obj} at {final_pos}")

    # Update robot position to the final object position
    robot_position = final_pos

# After all objects are placed, return to the initial position (0,0)
robot_actions.append(f"Move from {robot_position} to (0, 0)")

# Print robot action sequence
print("\nRobot Action Sequence:")
for action in robot_actions:
    print(action)
```

The robot's initial position can be set. For our case, we set it to (0,0), but we can determine this by taking in the data of the hand position of the video. We need to "move" from the robot's initial position to the next object's initial position. Then, the pinching phase has started, so the robot knows it needs to "pick" up the object. With the picked up object, the robot needs to move to the final position of the object, which is determined by the final frame of the pinching phase. In knowing this, the robot needs to "Place" the object at the final position of the object. The robot repeats this until all objects have been moved to its correct position in the video. As a final measure, the robot moves to its initial position should it need to do another task again. The output of the action sequence is shown below in Figure X.

```
Robot Action Sequence:
Move from (0, 0) to (634.5, 884.0)
Pick fork at (634.5, 884.0)
Move fork from (634.5, 884.0) to (1063.0, 454.0)
Place fork at (1063.0, 454.0)
Move from (1063.0, 454.0) to (909.5, 828.0)
Pick bowl at (909.5, 828.0)
Move bowl from (909.5, 828.0) to (743.5, 434.0)
Place bowl at (743.5, 434.0)
Move from (743.5, 434.0) to (504.5, 825.5)
Pick knife at (504.5, 825.5)
Move knife from (504.5, 825.5) to (1051.0, 392.0)
Place knife at (1051.0, 392.0)
Move from (1051.0, 392.0) to (501.5, 862.5)
Pick spoon at (501.5, 862.5)
Move spoon from (501.5, 862.5) to (247.0, 618.5)
Place spoon at (247.0, 618.5)
Move from (247.0, 618.5) to (0, 0)
```

Figure 19: Robot action sequence needed to perform the video demonstration.

The limitations of our approach is if the object cannot be detected correctly by the model used for object detection. The csv file is updated per frame according to which object is detected. In our case, we would sometimes detect 2 knives, which could skew some of the positions. The hand also covers the object, so that can potentially cause issues. We also used a camera from the topside, so we can utilize a 2D plane instead of a 3D plane in determining the movement of the objects according to the pixels of the camera.