Toronto Metropolitan University
Department of Mechanical, Industrial, and Mechatronics Engineering
Faculty of Engineering & Architectural Science

February 2025

ASL Robotic Hand Report

Project

Full ASL Robotic Hand Report + Bill of materials

Ammar J Mahmood

Table of contents:

1. Executive summary:

The purpose of this project is to bridge the gap between the hard of hearing and the deaf. This report outlines the work done by our team to create, train, and practically use a Computer Vision (CV) algorithm to detect ASL letters, translate them, and then have a robotic arm articulate them - all in real time. This project is an ambitious one that aims to utilize a fully functional CV algorithm, and a fully functional robotic hand (that can articulate each joint), and have these two systems run locally on a Raspberry Pi 5. This report will detail the project roadmap, the synthesis of the CV algorithm, the fabrication and construction of the robotic hand, and the marriage of these two systems and how they communicate.

Some challenges that this project presented were the training and accuracy of the CV algorithm, the cost and development time of the hand, and the off-loading of the project from a PC to a Raspberry PI. These problems were all met and solved through the iterative development detailed in this report; one example of how accuracy was increased is the creation of a large dataset for the CV algorithm. The robotic hand took time to develop; the 2 main parts of development were the building of the hand, and then programming it. The four milestones of assemblage development were the 3D printing of parts, the assembly of these parts, and the integration of the servo motors. The integration of the CV algorithm and the robotic hand was a process that the team planned for and worked towards (detailed in the timeline). One example of an issue encountered was a hit in performance when trying to run these processes; this issue was amended by implementing threads (see slide deck 5 on parallel computations from the course shell).

The key achievement of this project is the creation of a cohesive solution that runs locally on a Raspberry Pi 5 that's accurate and robust. The benefits of this project are the application of concepts taught in the MTE301 (note parallel computation and Object-Oriented Programming) and the exploration of new concepts, systems, and code bases such as OpenCV, MediaPipe, Raspberry Pi, and numerous other libraries and practices used in the training of CV algorithms and when working with the Pi. This project not only benefited the team academically but also aims to be of service to the public by working on a project that aims to aid marginalized communities.

2. Introduction:

Problem Statement:
The deaf and hard of hearing community faces significant communication barriers in everyday interactions with people who don't know American Sign Language (ASL). 2.80% of the USA's population uses ASL as a means of communication (reported by the National Library of Medicine [1]). This language barrier is further exemplified by the fact that only 37,620 Canadians reported to be able to converse in ASL; when considering the Canadian population of 40.1 million approximately 0.09% of the population can converse in ASL [2]. The majority of the US and Canadian population cannot understand or converse in sign language.

This creates a significant communication gap that impacts education, healthcare, employment, and social interactions. Current technological solutions primarily focus on one-way translation (converting ASL to text/speech) but don't address the need for non-ASL users to communicate back in sign language. Additionally, existing solutions often require specialized equipment or extensive training, making them impractical for widespread adoption.

The proposed solution addresses this bidirectional communication challenge through an innovative dual-mode system.
In one direction:

- Users can type letters or words that the robotic hand can physically articulate in ASL, allowing non-ASL users to communicate with the deaf community.

In the other direction:

- The system uses computer vision to recognize ASL letters performed by a user and convert them to text, enabling deaf individuals to communicate with those who don't understand sign language.

This two-way approach promotes inclusion by bridging this gap in communication, where both parties can interact using their preferred method of communication - either traditional text or ASL. The system's ability to both interpret and demonstrate ASL makes it particularly valuable as both a communication tool and a potential learning aid for those wanting to learn sign language.

Background:
American Sign Language (ASL) is a complete, natural language that uses signs made by moving the hands coupled with facial expressions and postures of the body. It is the primary language for many North Americans who are deaf and is one of several sign language options used by people who are hard of hearing. While previous technological solutions have focused on translating ASL to text or speech, our project takes the novel approach of creating a physical interface that can interpret and reproduce ASL letters, enabling two-way communication.

The scope of this project is to create a computer vision algorithm that takes in ASL letters from a user via their webcam, outputs the translation, and interpolates that data into something that the robotic hand can use to output ASL letters via that visual information. The intended outcome is to bridge the communication gap between people of varying capabilities. Our team used existing code bases and resources to aid us in this endeavour as much work has been done on joint tracking and computer vision algorithms.

The system utilizes a Raspberry Pi 5 as the main processing unit, leveraging its enhanced computational capabilities for real-time computer vision processing. The mechanical design incorporates servo motors and a custom 3D-printed hand structure, allowing for precise control of individual finger movements necessary for accurate ASL letter formation. This report will detail the development process, including the computer vision

implementation using Mediapipe and OpenCV, the mechanical design and control system of the robotic hand, and the integration of these components into a cohesive system.

Objectives:

- Implement a computer vision algorithm to recognize ASL letters via camera
- Develop real-time ASL letter recognition and translation to text
- Integrate hardware and software into a functional two-way communication system
- Design and build a robotic hand capable of demonstrating ASL letters
- Ensure reliable and accurate finger positioning through proper calibration and mapping
- Create a system that converts typed text into ASL letter demonstrations

System/Product/Process Description:

The final product at a high level is a computer vision algorithm that can track a hand's joint data in real-time, use that joint data to translate ASL letters from the computer's camera, and interpolate that data into commands that the robotic hand can use to articulate ASL letters in real-time. It is also important to note that the robotic hand is capable of taking input via the keyboard to output sign language too.

The computer vision algorithm uses OpenCV and Mediapipe to track hand joint positions in real time; this capability is utilized in 2 main ways:

1. The training of the algorithm and assembly of the database
    a. We take in 100 images of each ASL letter (23 letters so 2300 images total)
    b. Run these images through our "creatdataset.py" program to map the joint data and create the vector data for each letter
    c. This vector data is given to the classifier trainer which would then output a score of how much was formatted correctly
2. Live joint tracking that makes use of the database
    a. This formatted data is then used by the "inference_classifier.py" program to run the algorithm

The following flowchart details the training of the algorithm and its deployment:

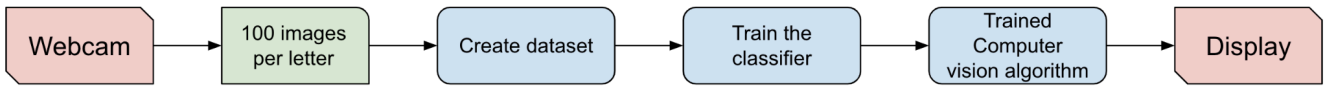


Figure 1: This flowchart details the process and programs used to train and run the CV algorithm. Red blocks denote hardware, blue blocks represent code files, and green blocks denote user input

The computer vision algorithm runs locally on the Raspberry Pi 5 which is also connected to the robotic hand. The live joint data is run through the computer vision algorithm, which sends purely binary data to the hand that signals to output a particular letter (not the mirroring of joint data). The robotic hand uses Python to wrap the following C++ files:

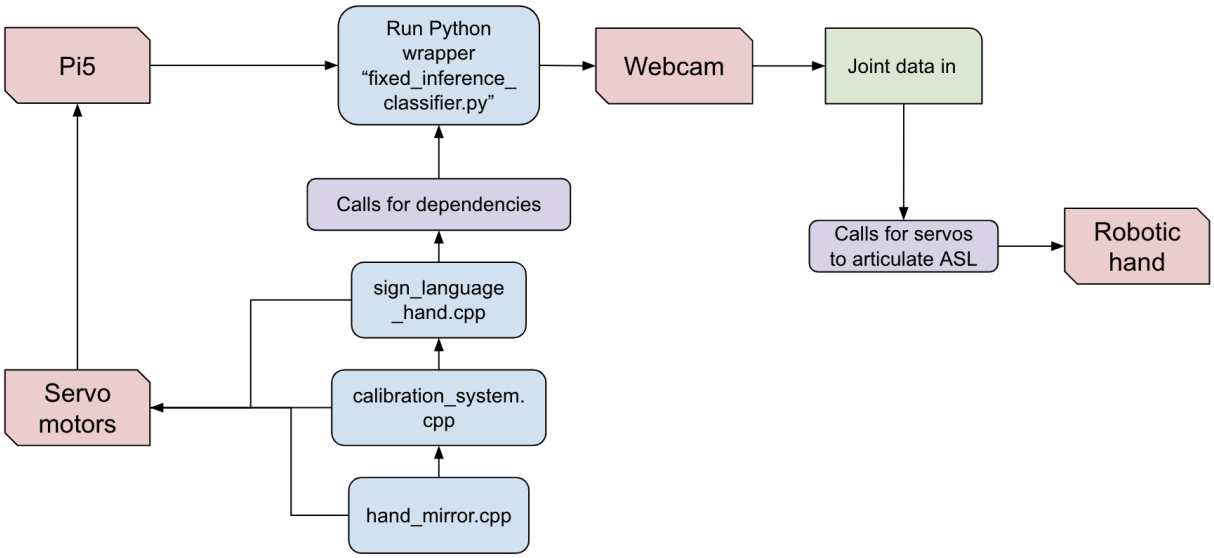


Figure 2: These files serve as the basis for how the hand articulates ASL letters and how it receives commands. Red blocks denote hardware, blue blocks denote code, green blocks denote user input and machine outputs, and purple blocks denote data transmission and the bridging of processes between hardware.

The following flowchart will outline the connections between the 2 main blocks of the project (CV algorithm and robotic hand):

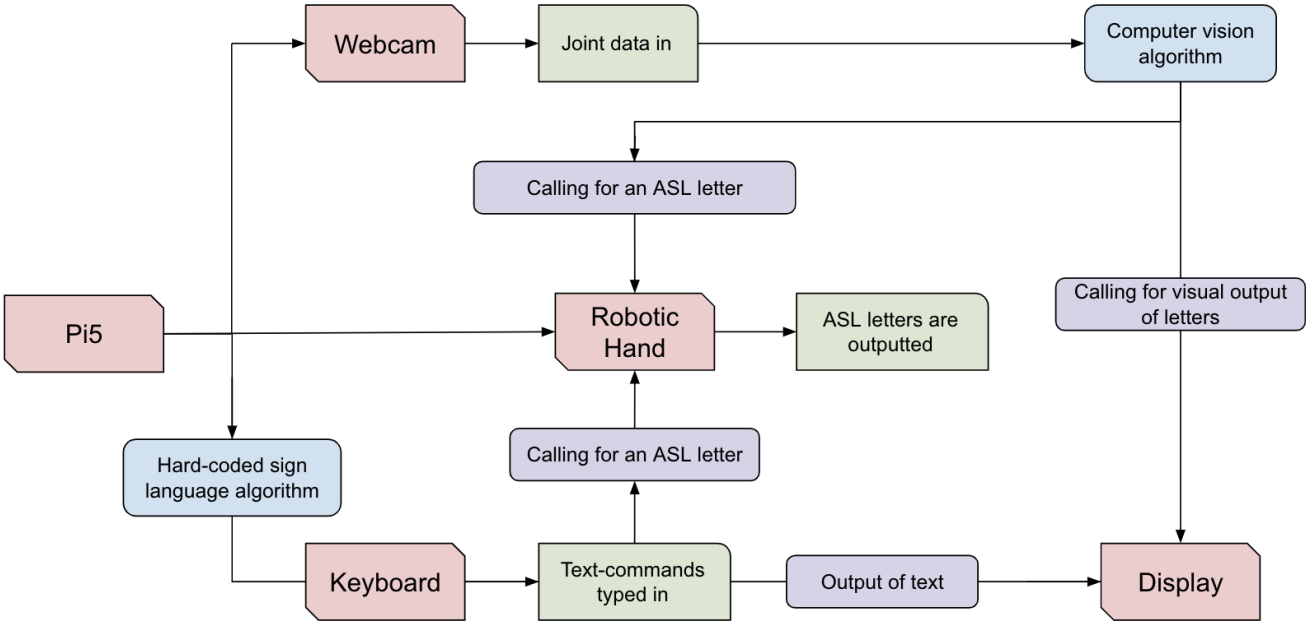


Figure 3: This flowchart details the entire process of the CV program, its outputs, and how it connects to the robotic hands. Red blocks denote hardware, blue blocks denote code, green blocks denote user input and machine outputs, and purple blocks denote data transmission and the bridging of processes between hardware.

3. Software Solution Design:

The software implementation of this robotic sign language system consists of multiple interconnected components, with Python serving as the high-level control system and C++ handling the low-level servo motor control. The primary Python program, fixed_inference_classifier.py, acts as the main controller, utilizing OpenCV and MediaPipe libraries for computer vision and hand tracking. This program processes real-time video input, detecting and analyzing hand gestures through MediaPipe's hand landmark detection system. The program leverages a pre-trained machine learning model (created using scikit-learn) to classify the detected hand gestures into corresponding ASL letters.

The system's hardware control is managed through several C++ programs, with sign_language_hand.cpp serving as the core implementation for controlling the servo motors. This program interfaces directly with the PCA9685 servo controller board through I2C communication on the Raspberry Pi 5. To ensure precise and calibrated movements, calibration_system.cpp provides a utility for fine-tuning the servo positions for each letter. The integration between Python and C++ is achieved through subprocess calls, where the Python program executes the compiled C++ programs with specific letter commands. Additional support files include hand_mirror.cpp for real-time mimicking of hand movements, and various utility programs for testing and debugging purposes. The entire system relies on several key dependencies, including OpenCV for image processing, MediaPipe for hand tracking, NumPy for numerical computations, and the Linux I2C libraries for hardware communication. This architecture allows for efficient processing of visual input while maintaining precise control over the mechanical components of the robotic hand.

About the code:

To start, the dependencies the machine learning program depends on are the following:

Python Dependencies:

1. Primary Libraries:
    - OpenCV (openCV-python version 4.7.0.68) - For computer vision and image processing
    - MediaPipe (version ≥0.10.0) - For hand tracking and landmark detection
    - NumPy - For numerical computations and array operations
    - scikit-learn (version ≥1.3.0) - For machine learning model implementation
2. Secondary Library
    - pickle - For model serialization
    - threading - For processing happening at the same time
    - queue - For thread-safe data exchange
    - pynput - For keyboard input handling
    - smbus - For I2C communication with the servo controller

C++ Dependencies:

1. Primary specific raspi libraries:
    - linux/i2c-dev.h - For I2C communication
    - fcntl.h - For file control
    - sys/ioctl.h - For I/O control
    - unistd.h - For POSIX operating system API
2. Standard Libraries:
    - iostream - For input/output operations
    - vector - For dynamic arrays
    - string - For string handling
    - map - For key-value pair storage
    - stdexcept - For exception handling
    - fstream - For file operations

- sstream - For string stream processing
- filesystem - For file system operations
- cstring - For C-style string manipulation
- memory - For smart pointer implementation

3. Hardware-Specific:
    - gpiod - For GPIO control on Raspberry Pi
    - PCA9685 - Servo controller driver implementation

Testing and Debugging:

4 main files are used to run the majority of the project:

1. fixed_inference_classifier.py
    a. Would take pictures of letters with “take_letter_pics_under_100.py”
    b. Then create dataset would be run
        i. create_dataset.py → data.pickle
    c. Then data.pickle would be used in “train_classifier.py” outputting “model.p”
    d. “model.p” would then be used in fixed_inference_classifier.py to classify the letter
    e. C++ execution files would be executed when a letter is detected “sign_language_hand_from_calibration.cpp”
2. sign_language_hand_from_calibration.cpp
    a. Runs letters based off a pickle file on the preferred degrees and angles for each letter and each finger from program “calibration_system.cpp”
3. hand_mirror.cpp
    a. Mirrors the hand detected in the camera
4. calibration_system.cpp
    a. Calibrate the fingers for “sign_language_hand_from_calibration.cpp”

Machine Learning Model Training:
The training of the ASL letters was done via a classical Machine Learning (ML) model approach that employs the use of large datasets that are then sorted based on some set decision tree. The data that was used came from vector data that MediaPipe can assign to human joints; this joint data serves as the base that the ML model is built on, this vector data assembled is one of the routes that the decision tree can take. Furthermore, the aforementioned data that is garnered comes from 100 images of each prospective letter that the ML model will encounter, these 100 images of the same letter will each have their own set of vector data that when arranged on a graph will appear in some sort of scatter plot. The reason for the creation of this large dataset is to normalize it (subtracting the minimum and maximum X and Y coordinates from each landmark) which helps improve the accuracy of the ML model, the larger the dataset, the more normal and better representative of the average result may look like.

Using the letter A as an example, the rest of this will run through how the files "create_dataset.py", "train_clasifier.py", and "inference_classifier.py" programs would sort the letter A's data. The "create_dataset.py" program is what does the post-processing of the images taken. It does this by calling on a built-in function from MediaPipe called "multi_hand_landmarks". This enables the use of the landmark data which is then run through a program that takes in all that landmark data for each joint in the hand and finally appends that to an array that will represent that letter - effectively creating the dataset. The following code snippets outline how data is taken in by the algorithm and how it is saved to an array:

*CODE SNIPPET FROM FILE "*create_dataset.py*"*

```
if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
              # Get x and y lists for normalization
                    x_ = []
                    y_ = []
                    for i in range(len(hand_landmarks.landmark)):
                        x = hand_landmarks.landmark[i].x
                        y = hand_landmarks.landmark[i].y
                        x_.append(x)
                        y_.append(y)
...
# Convert to numpy arrays and check shapes
data = np.array(data)
labels = np.array(labels)

print(f"Data shape: {data.shape}")
print(f"Labels shape: {labels.shape}")
print(f"Total processed images: {len(data)}")
print(f"Number of classes: {len(set(labels))}")

# Save the processed data
f = open('data.pickle', 'wb')
pickle.dump({'data': data, 'labels': labels}, f)
f.close()
```

```
print("Data saved to data.pickle")


—-----------------------------------------------------------------

data: [0.17412111163139343, 0.6576124876737595, 0.24597910046577454,
0.5176725834608078, 0.22806480526924133, 0.344136580824852, 0.16925987601280212,
0.22980497777462006, 0.12035289406776428, 0.16221053898334503,
0.1792522370815277, 0.30046533048152924, 0.17192164063453674,
0.1661376804113388, 0.1642179787158966, 0.0892198234796524, 0.15697231888771057,
0.016467615962028503, 0.12001007795333862, 0.3139815777540207,
0.0806572437286377, 0.17512325942516327, 0.054185956716537476,
0.08083154261112213, 0.03233063220977783, 0.0, 0.06653392314910889,
0.35733388364315033, 0.04545280337333679, 0.2612334042787552,
0.09644612669944763, 0.3395148366689682, 0.1273692548274994, 0.4122585505247116,
0.014518022537231445, 0.41592954099178314, 0.0, 0.32042525708675385,
0.04839998483657837, 0.36667971312999725, 0.08475100994110107,
0.4248254746198654]

This is x[] value
[0.5355375409126282, 0.6073955297470093, 0.5894812345504761, 0.5306763052940369,
0.481769323348999, 0.5406686663627625, 0.5333380699157715, 0.5256344079971313,
0.5183887481689453, 0.48142650723457336, 0.44207367300987244,
0.4156023859977722, 0.3937470614910126, 0.42795035243034363,
0.40686923265457153, 0.4578625559806824, 0.48878568410873413,
0.3759344518184662, 0.36141642928123474, 0.4098164141178131, 0.4461674392223358]

This is min x[] value 0.36141642928123474 This is x value 0.4461674392223358
normalize = 0.4461674392223358 - 0.36141642928123474
```

Figure 1: This block of vectors is what the raw landmark data looks like pre processing and post processing

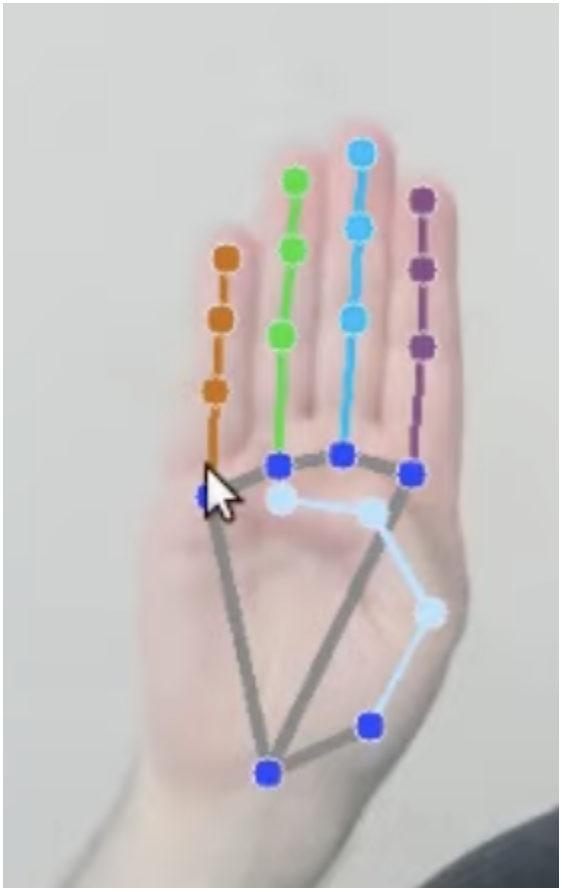


Figure 2: Images that use the “draw_landmarks” to output images with the landmark data visually represented

With the image now translated into usable data (in the form of an array), this data is handed off to the “train_clasifier.py” program. This data needs to be prepared in a format such that an ML model can be trained to detect the letter A. This file can do this by splitting the data into a training set and a test set. The training set (denoted by “x_train” and “y_train” in the code) is used by the Random Forest Classifier to assemble this model. The testing set (denoted by “x_test” and “y_test” in the code) is not touched until the end of the program and is used to validate the accuracy of the training set per the Random Forest Classifier model. This data is then tested for accuracy by comparing the predicted data outputs (per the training model) against the y_test (labels) to output the model’s percentage of success with classifying all the data. The Random Forest Classifier was chosen because of its robustness and ability to handle large decision trees. The explanation as to how such a model functions is far beyond the scope of what is relevant to this project, and will not be explained further. The following code snippets outline how the model was made (testing and training set) and the creation of model.p:

*CODE SNIPPET FROM FILE “*train_clasifier.py*”*

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2,
shuffle=True, stratify=labels)

model = RandomForestClassifier()

model.fit(x_train, y_train)
```

```
y_predict = model.predict(x_test)

score = accuracy_score(y_predict, y_test)

print('{}% of samples were classified correctly !'.format(score * 100))

f = open('model.p', 'wb')
pickle.dump({'model': model}, f)
f.close()
```

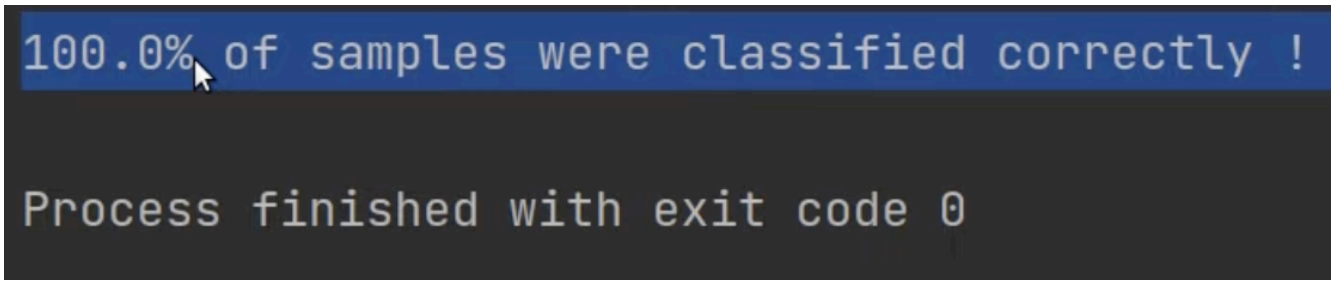


Figure 3: Upon completion of the ML model, this message will be printed.

With the model assembled, the final file “inference_classifier.py” uses these formatted datasets (denoted by filename “model.p”) and finally labels all the letters from 0 to 24 and runs it through a for loop that searches to see if the live feed of the user’s hand matches data from “model.p” and then creates a box around the user’s hand with what letter it is detecting in the top left corner of the box. The following code denotes the labelling of the dataset, and the framing:

CODE SNIPPET FROM FILE

```
labels_dict = {
    0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F',
    6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K', 11: 'L',
    12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R',
    18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W', 23: 'X',
    24: 'Y'  # Note: Z is typically not included as it requires motion
}
while True:
    data_aux = []
    x_ = []
    y_ = []

    ret, frame = cap.read()
    if not ret:
        continue
```

```
    H, W, _ = frame.shape
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    results = hands.process(frame_rgb)
    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(
                frame,
                hand_landmarks,
                mp_hands.HAND_CONNECTIONS,
                mp_drawing_styles.get_default_hand_landmarks_style(),
                mp_drawing_styles.get_default_hand_connections_style())

            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y
                x_.append(x)
                y_.append(y)

            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y
                data_aux.append(x - min(x_))
                data_aux.append(y - min(y_))

        x1 = int(min(x_) * W) - 10
        y1 = int(min(y_) * H) - 10
        x2 = int(max(x_) * W) - 10
        y2 = int(max(y_) * H) - 10

        prediction = model.predict([np.asarray(data_aux)])
        predicted_character = labels_dict[int(prediction[0])]

        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
        cv2.putText(frame, predicted_character, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0, 0), 3,
                    cv2.LINE_AA)

    cv2.imshow('frame', frame)
    if cv2.waitKey(1) & 0xFF == 27:  # Press 'Esc' to exit
        break
cap.release()
cv2.destroyAllWindows()
```

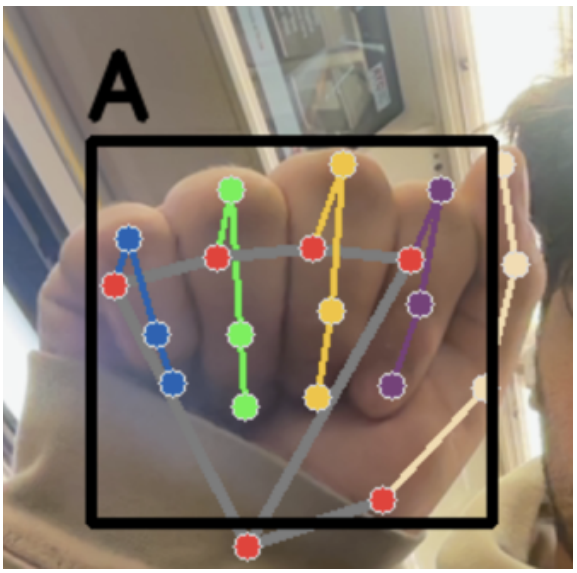


Figure 4: This is an example of what the finished ML model would output when detecting ASL letters.

Version and Iteration of Python ML code:

Version 1:

The first iteration of the sign language detection system faced significant compatibility issues. The primary challenge arose from version mismatches between Python and OpenCV when running on Ubuntu. This version attempted to implement basic hand detection but proved unstable due to these fundamental compatibility conflicts. The code structure was basic, focusing solely on detecting hand positions with hard-coded joint positions without robust error handling or optimization.

Version 2:

To address the compatibility issues, a Miniconda virtual environment was implemented (on the Raspberry Pi), ensuring consistent package versions across the system. However, this version revealed a critical flaw in the detection algorithm - the system would frequently misidentify background objects as hands, causing program crashes. The detection was overly sensitive, resulting in false positives that significantly impacted system reliability and overall performance.

Version 3:

The third version marked a significant improvement in detection accuracy. Focused object tracking was implemented, prioritizing maintaining the detection of an initially identified hand effectively reducing background interference. Key improvements included:

- Implementation of hand-tracking persistence
- Background noise filtration
- Optimization for dual-hand detection while maintaining priority on the primary hand
- Enhanced error handling for detection losses

Version 4:
This iteration introduced several features while addressing performance issues:

- Implementation of word spelling functionality
- Addition of text-based input/output interface
- Camera lag compensation through frame buffer optimization
- Introduction of a two-second detection delay to prevent rapid-fire letter recognition

Performance Optimization To address the camera lag issues on the Raspberry Pi, we implemented several optimizations:

- Reduced frame processing resolution while maintaining detection accuracy
- Implemented frame skipping when processing burden exceeded capacity
- Added frame buffering to smooth out processing peaks
- Optimized memory usage through efficient data structure management

Control Flow Integration The final version implemented a sophisticated control flow system:

- Implemented queue-based letter processing
- Added state management for robot movement completion
- Created handoff protocols between detection and execution phases
- Integrated feedback loops for movement confirmation

The system now maintains a stable pipeline from detection through execution, managing the transition between identified letters and physical movements effectively. The two-second delay between detections provides crucial stability, allowing the robotic hand to complete each letter formation before initiating the next movement.

Reflection:
Some problems were encountered with lag on the video feed and the 8 servos running at the same time as the machine learning model. However, the Raspberry Pi 5 came equipped with 8GB RAM as well as an ARM Cortex quad-core processor, thus providing 4 physical CPU cores capable of handling 8 concurrent threads through simultaneous multithreading.

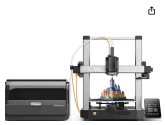
The solution was realized during the final iterations of the code from the final lessons in class. The threading capability is particularly advantageous for the ASL robotic hand project, where multiple operations need to run concurrently. The system has three primary threads: the camera thread for continuous video capture and hand gesture recognition, the input thread for handling user text input, and the prediction thread for processing detected gestures and controlling the servo motors.

4. Hardware Solution Design:

Bill of Materials:

Item Image	Item Name	Quantity	Price	Link to item

	OV2643 USB Camera Module Autofocus Mini Camera Board 2MP 120° Wide Angle Camera Module with OV2643 Chip	1	$28.77	Amazon link to camera
	iRasptek Starter Kit for Raspberry Pi 5 RAM 8GB - 128GB Edition of OS-Bookworm Pre-Installed (Aluminum Case)	1	$225.99	Amazon link to raspberry pi 5 8gb ram with SD card and full kit
	Besomi Electronics PCA9685: 16-Channel 12-Bit PWM/Servo Driver Module with I2C Interface - Precise Control for Robotics, Automation, and IoT Projects	1	$12.71	Amazon link to servo driver module
	DC Power Supply Variable, 30V 10A Adjustable Switching Regulated DC Bench Power Supply with High Precision 4-Digits LED Display, 5V/2A USB Port, Coarse and Fine Adjustments Jesverty SPS-3010N	1	$76.49	Amazon link to power bench supply (must be at least 10A and 30V preferably if running servos all at same time)
	6-Pack MG996R Servo,Aideepen Metal Gear High Speed Torque Servo Digital Servo Motor for Smart Car/Robot/Boat, Helicopter,DIY	2 (only 8 used total of 12)	$54.23 x 2 = $108.46	Amazon link to MG996R servos (can use any other ones as well from Tower pro, non continuous servos, check torque weight if you wanna lift things)
	Trilene Big Game Clear 0.024in | 0.60mm	1	$19.07	Amazon link to nylon fishing string
	ANYCUBIC High Speed 3D Printer Filament 1.75mm, Print Up to 10X Faster, Rapid PLA Filament with High Prints Quality, Dimensional Accuracy +/- 0.02mm, Print with Most FDM 3D Printers, 1KG Spool, Black	2	$25.98	Amazon link to PLA for printer anycubic (or any printer)
	120pcs Multicolored Dupont Wire 40pin Male to Female, 40pin Male to Male, 40pin Female to Female Breadboard Jumper Wires Ribbon Cables	1	$9.99	Amazon link to wires used
OPTIONAL (if unavailable, some come with other parts and are not necessary for the project)				

	2Pcs 4MM Banana Plug to Alligator Clips 15A Test Lead Wire Cable Set 14AWG for Multimeter Oscilloscope Testing Probe 1M	1	$5.99	Amazon link to banana plug if does not come with ur power bench supply (mine did and I bought by accident)
	20 Gauge Wire 2 Conductor Electrical Wire, 20 AWG Wire Stranded PVC Cord, 12V Low Voltage/Tinned Copper/Flexible/20/2 Wire for Automotive Wire LED Strips Lamp Lighting Marine (30FT-9.1M)	1	$15.99	Amazon link to gauge wire if you don't have male and female jumper cables
	10pcs/set, 60W soldering iron kit, soldering gun, 110V soldering iron kit tool, temper	1	$11.75	TEMU or any place you want however is not necessary for the project
	ANYCUBIC Kobra 3 Combo 3D Printer Smart Multicolor Printing (4 to 8 Colors), 600 mm/s High Speed FDM 3D Printer, LeviQ 3.0 Auto-Leveling, RFID Sync, 250 * 250 * 260mm Larger Printing Size	1	$599.00	Amazon link to 3d printer
	Bambu P1S	1	$949.00	Bambu P1s (on sale right now)
	CYAFIXED Super Glue Gel, All-Purpose Superglue, Cyanoacrylate Instant Adhesive for Plastic, Wood, Metal, Repair - Four 3 Gram Tubes, Clear	1	$7.19	Amazon link to super glue or from dollarama
TOTAL			$507.46 ⇒ (if optional items are purchased ⇒ $1,147.38)	

Robotic Hand Timeline:

The development of the robotic hand was a challenging and rewarding journey that spanned approximately 4 weeks. Development began with the designing and 3D printing of the forearm and fingers,[5] which took four days to complete and used two spools of PLA. While waiting for the servo motors to arrive, assembly of the printed parts (forearm and fingers)[5] was done. Threading the nylon wire through each finger was a process that required precision and accuracy; this took three days. Installing the fingers into the servos presented another challenge, taking two days due to the lack of correct screws and screwdriver bits.

The first iteration of the code was written for an Arduino and the hardware was configured accordingly. However, the project later transitioned to a Raspberry Pi 5 but used the Arduino servo shield. During this process, a low power issue was encountered (as a result of running 8 servos and the CV algorithm), and the purchase of a bench power supply was necessary to solve it. Once resolved, the code was tested again, and it worked successfully.

Finally, the base was designed and printed; this took another four days due to its size. Attaching the base and implementing the waving motion required a full day of effort. Wiring proved to be time-consuming, as the team had limited experience programming with a Raspberry Pi and managing circuitry. This project required extensive time spent reading documentation, testing individual servos, and running and testing code to ensure proper functionality.

A variety of software was used during the project such as Bambu Slicer, Prusa Slicer, VsCode, Geany, G++, RaspiOS, Solidworks, Tinkercad and Fusion360.

**See instructables reference number 5 for parts and google drive for custom parts

Testing and Debugging:

1. First Iteration:
    - Designed and 3D printed the forearm and fingers
    - Assembled the printed parts and attached the fingers to the forearm
    - Threaded nylon wire through each finger
    - Installed fingers into the servos
    - Wrote code for an Arduino and wired the hardware accordingly
    - Encountered power issues and purchased a bench power supply
    - Tested the code successfully
2. Second Iteration:
    - Transitioned from Arduino to Raspberry Pi 5
    - Used an Arduino servo shield
    - Designed and printed additional parts for the base
    - Attached the base and implemented the waving motion
    - Spent time reading documentation and testing each servo individually
    - Ran the code to ensure proper functionality
3. Third Iteration:
    - Designed new parts for the joints to improve fit
    - Attempted to integrate a gear system for the wrist servo but encountered attachment issues
    - Connected the Raspberry Pi to a PCA board instead of a servo shield
    - Attached a camera Camera Module to the Pi
    - Explored alternative communication methods to reduce reliance on Wi-Fi, such as implementing SSH for remote access and having the Pi emit its own Wi-Fi network for offline connectivity

Version and Iteration of C++ servo code:

Version 1:

This part of the project started with an attempt to use the GPIO Zero library for direct servo control. This approach was straightforward in concept but ultimately proved unworkable with the Raspberry Pi 5. A compatibility issue was discovered early in development when attempting to implement basic servo control functions.

Version 2:

In this version, the libgpiod library was used (as demonstrated in servocpp1.cpp). This version successfully controlled a single servo connected to GPIO pin 18. The code implemented a Servo class that handled pulse width modulation (PWM) signal generation with precise timing control. For example, the "setAngle" function converted angle values (0-180 degrees) into appropriate pulse widths:

```
void setAngle(int angle) {
    const auto cycleTime = 20000us;  // 20ms cycle (50Hz)
    const auto pulseWidth = microseconds(500 + (angle * 2000 / 180));
}
```

Version 3:

This version marked a significant development in the introduction of the PCA9685 servo controller board, as shown in "movefingers.cpp". This version established I2C communication and implemented proper register control for a single servo. The code included essential initialization sequences and PWM frequency settings:

```
static constexpr int I2C_ADDR = 0x40;
static constexpr uint8_t MODE1 = 0x00;
static constexpr uint8_t PRESCALE = 0xFE;
writeRegister(MODE1, 0x00);
setFrequency(50);  // 50Hz for servos
```

Version 4

This version is represented by "aslletters.cpp" and later refined in "sign_language_hand.cpp", implementing full control of all five servos with predefined positions for ASL letters. This version introduced a structured approach to letter configurations:

```
letterConfigs['A'] = FingerPosition(
    {FINGER_HALF_BENT, FINGER_BENT, FINGER_BENT, FINGER_BENT, FINGER_BENT},
    "Fist with thumb resting on side"
);
```

The version also incorporated calibration capabilities through a configuration file system, allowing for fine-tuning of servo positions for each letter. This ensured more accurate and consistent letter formations across different hardware setups. Each version of the code both in C++ and in Python represented a significant step forward in functionality and reliability, with the final version providing a robust foundation for the ASL robotic hand system.

Pseudocode for general C++ servo movement (with PCA board):

```
// Constants and Configuration
#define PCA9685_ADDRESS 0x40
#define MODE1_REGISTER 0x00
#define PRESCALE_REGISTER 0xFE
#define LED0_ON_L_REGISTER 0x06

// Servo position constants (PWM values)
#define SERVO_STRAIGHT 375    // Fully extended position (0 degrees)
#define SERVO_BENT 150       // Fully bent position (90 degrees)
#define SERVO_HALF_BENT 263  // Half bent position (45 degrees)

class SignLanguageHand {
    // Class member variables
    private:
        int i2c_fd;  // I2C file descriptor
        map<char, array<int, 5>> letterConfigurations;  // Store finger
positions for each letter

    // Initialization
    Initialize():
        Open I2C device
        Set up PCA9685 board
            Set sleep mode
            Set PWM frequency (50Hz)
            Wake up device
        Initialize letter configurations map
        Move to rest position

    // Configure all letter positions
    InitializeLetterConfigs():
        For each letter (A-Z, excluding J and Z):
            Store array of 5 servo positions (thumb, index, middle, ring, pinky)
            Store description of hand position
            Example for 'A':
                letterConfigs['A'] = {
                    SERVO_HALF_BENT,  // Thumb
                    SERVO_BENT,       // Index
                    SERVO_BENT,       // Middle
                    SERVO_BENT,       // Ring
                    SERVO_BENT        // Pinky
                }
```

```
    // Control individual servo
    SetServoPosition(channel, position):
        Calculate PWM values
        Write to PCA9685 registers
        Add small delay for smooth movement

    // Display letter

DisplayLetter(letter):
        Convert letter to uppercase
        If letter configuration exists:
            Get servo positions for letter
            For each finger (0 to 4):
                SetServoPosition(finger, positions[finger])
                Add small delay between fingers
        Else:
            Throw error for unsupported letter

    // Reset hand position
    ResetPosition():
        For each finger (0 to 4):
            SetServoPosition(finger, SERVO_STRAIGHT)
            Add small delay between fingers

    // Main program loop
    Main():
        Initialize hand
        If command line argument exists:
            DisplayLetter(argument)
        Else:
            Run test sequence:
                Test each finger individually
                Test a few sample letters
                Reset position
        Clean up and close I2C connection
```

Reflection:
The project was a valuable learning experience that enabled the team to overcome numerous challenges and create a functional robotic hand capable of replicating ASL letters by using problem-solving skills, and the lessons taught in the course. With each iteration, the design was refined, issues were addressed, and the final product was improved, thus increasing the overall functionality and robustness of the robotic hand and its software.

5. Conclusions:

Possible improvements/Future work:
The current system, while functional, has several areas where improvements could enhance its performance and usability:

1. The hand's mechanical design could be improved by:
    - Adding springs to finger joints to create more natural flexion and extension movements
    - Extending the range of motion in each finger through redesigned servo mounts
    - Incorporating additional joints in the thumb for more complex movements
    - Redesigning the palm to allow for more natural finger positioning
    - Implementing a modular design where components can be easily replaced or upgraded
    - Adding a cooling system for continuous operation during extended periods

2. The system's portability could be enhanced by:
    - Integrating a built-in rechargeable battery pack instead of relying on external power
    - Creating a more compact housing for all electronic components
    - Making the overall design more ergonomic and easier to carry
    - Reducing the number of exposed wires and connections
    - Adding a built-in display screen to eliminate the need for external monitors
    - Including LED indicators to show system status and letter recognition

3. The computer vision system's accuracy could be increased by:
    - Training the model with a larger dataset (the current model uses ~2,500 images)
    - Including more diverse lighting conditions in training data
    - Adding more hand orientations, angles, and positions to the dataset
    - Collecting data from users of different hand sizes and skin tones
        - Taking into account, hands of varying finger lengths can be useful
    - A potential way to get a variety of hand types, camera angles, and lighting conditions would be to completely digitize the collection of training data by using a virtual environment such as Unreal Engine 5 (UE5)
        - this would also allow for the collection of a much larger dataset at a much faster rate
4. Considering the nuances of language within ASL itself and the ASL community
    - When considering the imperfect medium of language there are nuances to consider such as sarcasm, slang, and short-form abbreviations of terminology
    - This is a problem that plagues all forms of communication and is not mutually exclusive to ASL
    - This is an important consideration for the potential future of this project and is relevant when considering the integration of words into the language model

These improvements would make the system more reliable, portable, robust, natural, and accurate in its movements while maintaining its core functionality as a communication tool. The addition of visual indicators, a cooling system, a modular design, and an integrated display would significantly enhance the user experience for non-experts and make the system practical for everyday users

6. Appendices:

Flowchart:
https://docs.google.com/drawings/d/1sblm0O3-8ReviLZ91nVVggTBDc7QIXKzEQOr-Lb64dI/edit

Code:
https://github.com/ammarjmahmood/robotic-asl-arm/tree/main
https://drive.google.com/drive/folders/1J2ZI0FKPKZaBmCE5oBdd9qllUZfgiPaF?usp=drive_link

Demonstration:
https://drive.google.com/file/d/1trR3abMgd3z0N62KdoEl0rV0joa-Oxul/view?usp=sharing
https://drive.google.com/file/d/1nGZxFWSxn7ltsSSAvxEMj0UJ33arZMPg/view?usp=sharing

Resources/Research:
[1] S. Bouzid et al., "Machine learning applied for sign language recognition: A comprehensive review," Med. Biol. Eng. Comput., vol. 61, no. 2, pp. 275-291, Feb. 2023, doi: 10.1007/s11517-022-02729-3.
[2] Statistics Canada. (2023, Sep. 23). "International Day of Sign Languages" [LinkedIn post]. Available: https://www.linkedin.com/posts/statcan_internationaldayofsignlanguages-asl-lsq-activity-7111348600944648192-E34N/
[3] "Sign Language Tutorial," YouTube, 2023. [Online]. Available: https://youtu.be/MJCSjXepaAM?si=XLZTz2dIUaPDMfEa
[4] Google. "MediaPipe Hands," MediaPipe Documentation, 2024. [Online]. Available: https://mediapipe.readthedocs.io/en/latest/solutions/hands.html
[5] Instructables. "ASL Robotic Hand (Left)," 2024. [Online]. Available: https://www.instructables.com/ASL-Robotic-Hand-Left/
[6] Raspberry Pi Foundation. "Physical Computing with Python," 2024. [Online]. Available: https://projects.raspberrypi.org/en/projects/physical-computing/1
[7] "GPIO Programming Discussion," Raspberry Pi Forums, 2024. [Online]. Available: https://forums.raspberrypi.com/viewtopic.php?t=327539
[8] Raspberry Pi Foundation. "GPIO in Python," Operating System Documentation, 2024. [Online]. Available: https://www.raspberrypi.com/documentation/computers/os.html#gpio-in-python
[9] "i2c-dev.h," GitHub Repository - Raspberry Pi Tools, 2024. [Online]. Available: https://github.com/raspberrypi/tools/blob/master/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/arm-linux-gnueabihf/libc/usr/include/linux/i2c-dev.h
[10] Raspberry Pi Foundation. "Raspberry Pi Tools," GitHub Repository, 2024. [Online]. Available: https://github.com/raspberrypi/tools/tree/master
[11] "Raspberry Pi Tutorial 1," YouTube, 2024. [Online]. Available: https://www.youtube.com/watch?v=i7CkBZVkD5E
[12] "Raspberry Pi Tutorial 2," YouTube, 2024. [Online]. Available: https://www.youtube.com/watch?v=9KkzekAA_NQ
[13] Raspberry Pi Foundation. "C SDK Documentation," 2024. [Online]. Available: https://www.raspberrypi.com/documentation/microcontrollers/c_sdk.html
[14] Google. "Hand Landmarker," MediaPipe Documentation, 2024. [Online]. Available: https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker
[15] Lingvano. "Sign Language Alphabet," ASL Blog, 2024. [Online]. Available: https://www.lingvano.com/asl/blog/sign-language-alphabet/

https://pubmed.ncbi.nlm.nih.gov/36423340/
https://www.linkedin.com/posts/statcan_internationaldayofsignlanguages-asl-lsq-activity-7111348600944648192-E34N/
https://youtu.be/MJCSjXepaAM?si=XLZTz2dIUaPDMfEa
https://mediapipe.readthedocs.io/en/latest/solutions/hands.html
https://www.instructables.com/ASL-Robotic-Hand-Left/
https://projects.raspberrypi.org/en/projects/physical-computing/1
https://forums.raspberrypi.com/viewtopic.php?t=327539
https://www.raspberrypi.com/documentation/computers/os.html#gpio-in-python
https://github.com/raspberrypi/tools/blob/master/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/arm-linux-gnueabihf/libc/usr/include/linux/i2c-dev.h
https://github.com/raspberrypi/tools/tree/master
https://www.youtube.com/watch?v=i7CkBZVkD5E
https://www.youtube.com/watch?v=9KkzekAA_NQ
https://www.raspberrypi.com/documentation/microcontrollers/c_sdk.html
https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker
https://www.lingvano.com/asl/blog/sign-language-alphabet/
https://drive.google.com/drive/folders/1J2ZI0FKPKZaBmCE5oBdd9qllUZfgiPaF?usp=drive_link