Bharat AI-SoC Student Challenge - Problem Statement 2

Touchless HCI for Media Control Using Hand Gestures on NVIDIA Jetson Nano

Project Report

Team Members: SriPrahlad Mukunthan
Rushil Jain
Shresh Parti

College: National Institute of Technology, Karnataka

College Mentor: Dr. Sumam David

1. System Architecture

1.1 Hardware Platform

The system is designed to run on the NVIDIA Jetson Orin Nano, a high-performance edge AI computing platform featuring:

- GPU: 512-core NVIDIA Ampere architecture with 16 Tensor Cores (providing up to 20 TOPS of AI performance)
- CPU: 6-core Arm Cortex -A78AE v8.2 64-bit CPU @ 1.5 GHz
- Memory: 4GB 64-bit LPDDR5 RAM with 34 GB/s bandwidth
- Video Capture: USB 3.2 Gen 2 (10Gbps) interface for high-bandwidth, real-time video capture via webcam

1.2 Software Stack

Component	Technology
Operating System	Jetpack OS
Programming Language	Python 3.10
Hand Detection	MediaPipe Hands v0.10 (Google)
Computer Vision	OpenCV 4.5+
Numerical Processing	NumPy 1.21+ (optimized linear algebra)
Input Simulation	pynput (keyboard control library)
System UI	tkinter
ML learning model	Scikit-learn (Random Forest Classifier)

1.3 Modular Design

The system follows a modular architecture with clear separation of concerns:

- **hand_tracker.py** - Interfaces with MediaPipe for 21-point hand landmark detection
- **utils.py** - Geometric calculations (distances, angles) using NumPy
- **gesture_recogniser.py** - Gesture classification logic
- **media_controller.py** - Keyboard command generation for VLC
- **config.py** - Centralized configuration and gesture mappings
- **main.py** - Integration and main control loop

- **app_ui.py -** Central graphical control hub, to launch the gesture recognition engine, and provide an interface for recording and training custom machine learning models
- **collect_gesture_data.py -** Capture and normalize landmark coordinates over 300 frames per gesture, saving the resulting dataset to a CSV file for training ML models
- **train_model.py -** Uses the samples collected to train the Random Forest Classifier Model and make a new gesture

2. Methodology

2.1 Preprocessing using OpenCV

The system captures the video in BGR format, which is converted to RGB format using OpenCV cvtColor command. This is done as mediapipe requires the video in RGB format.

2.2 Hand Landmark Detection

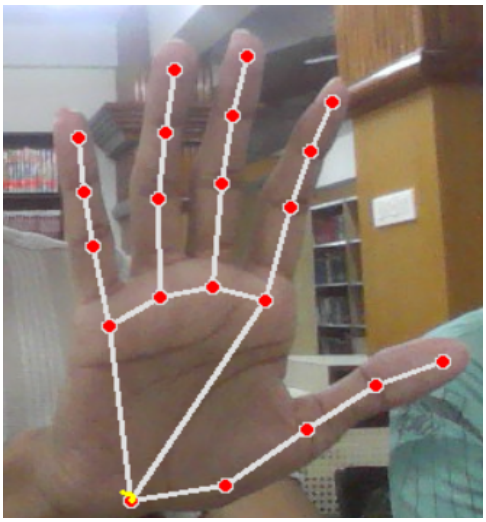
The system utilizes Google's MediaPipe Hands framework, which employs a two-stage pipeline:

Stage 1: Palm Detection

A lightweight neural network detects the palm region using a single-shot detector (SSD) architecture, returning a bounding box.

Stage 2: Hand Landmark Localization

A regression model predicts 21 3D hand keypoints (x, y, z coordinates) representing joints, fingertips, and the wrist. The z-coordinate is relative to the wrist, enabling depth-aware gesture detection.



Hand Landmarks detected using Mediapipe

The **get_landmarks** function in the **hand_tracker** module is used to acquire the normalised landmarks coordinates along with the landmark IDs, which is the basis of our rule based gesture detection. It also sends the hand label (whether left or right). The hand label is required for the dynamic pinch rotate gesture.

2.2 Gesture Recognition

2.2.1 List of gestures implemented

- Static
  - Open Palm
  - Fist
  - Pinch
  - Index finger extended
- Dynamic
  - Swipe left and right
  - Pinch rotate clockwise and anticlockwise
  - Fist up and down

2.2.2 Rule-Based Detection (Primary Method)

The system primarily employs geometric rule-based classification, chosen for its speed, interpretability, and suitability for distinct gestures. Key techniques include:

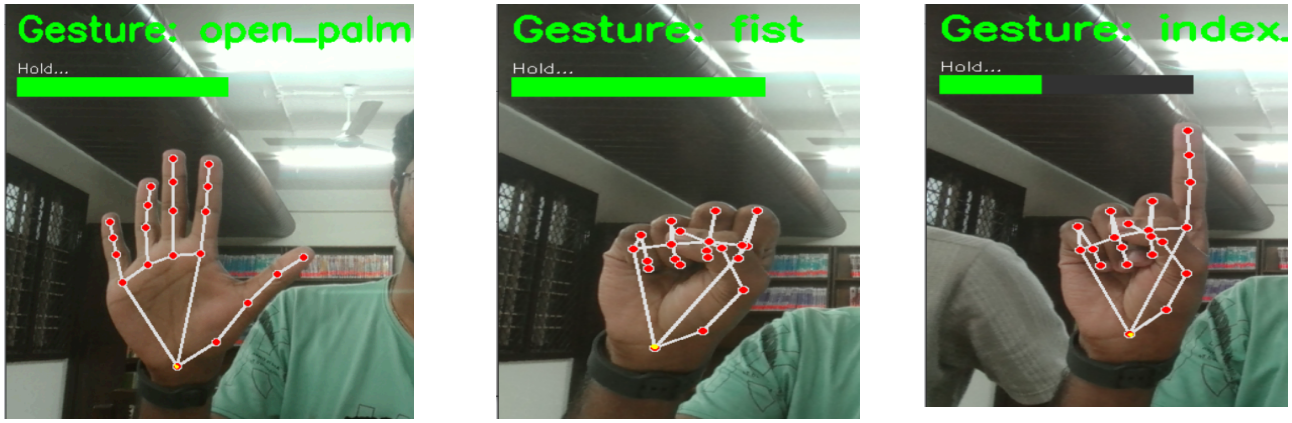
Static gesture detection:

Angle-Based Finger Extension Detection:

For each finger, the angle at the proximal interphalangeal (PIP) joint is calculated using the dot product of vectors.

Fingers with angles >160° are classified as extended. This method proved robust across hand orientations, eliminating the need for left/right hand classification.

The number and type of finger extended are used to detect **open palm** (all fingers extended), **fist** (all fingers closed), **index finger extended**.



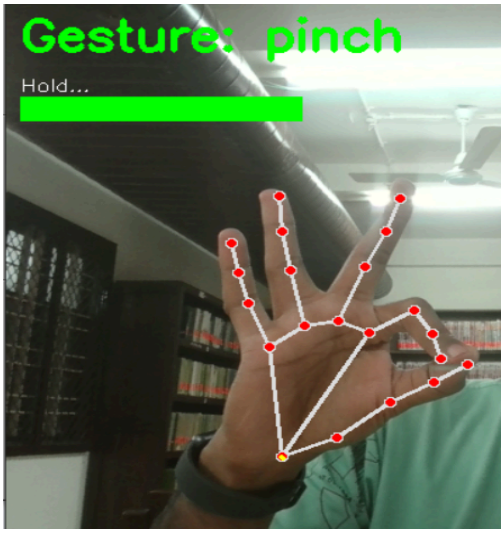
Open palm, fist and index finger detected

Ratio-Based Pinch Detection:

Pinch is detected using a scale-invariant ratio:

pinch_ratio = distance(thumb_tip, index_tip) / distance(wrist, middle_finger_tip)

A threshold of 0.25 (25% of hand length) proved optimal, making detection independent of camera distance.



Pinch detected

Dynamic gesture detection:

Swipe Detection:

Horizontal swipes are detected by tracking wrist position over 10 frames using a circular buffer (collections.deque). A swipe is registered when horizontal displacement exceeds 200 pixels with minimal vertical movement, following a 1.0-second settle period to prevent false triggers.

Vertical Fist Movement:

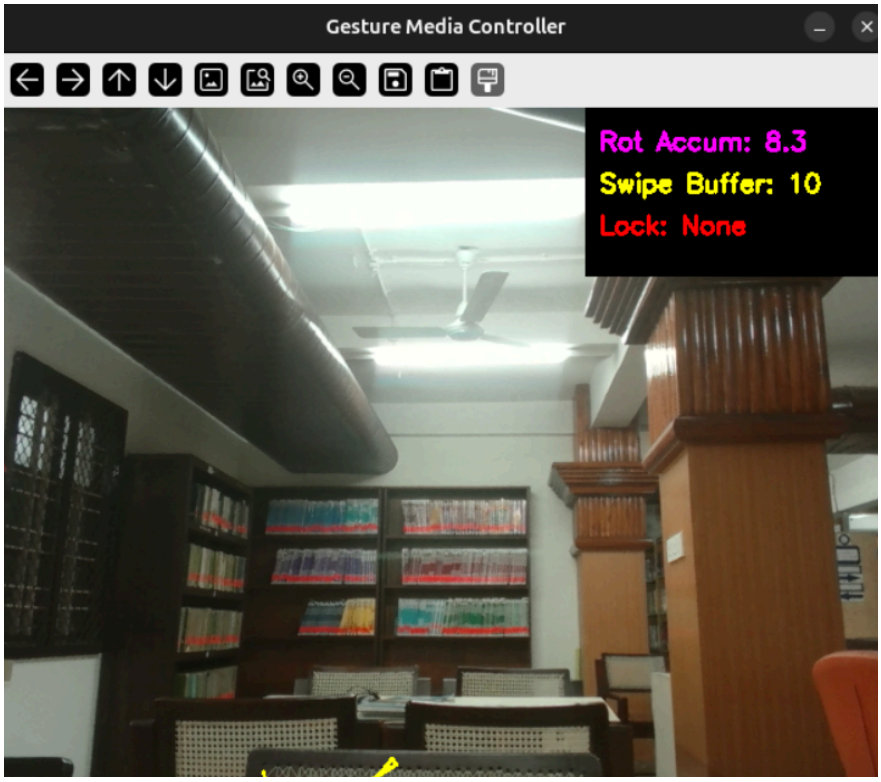
Vertical fist movements are detected by monitoring the wrist position of a closed fist over a 10-frame buffer. A Fist Up or Fist Down is registered when the vertical displacement exceeds a 150-pixel threshold, provided the movement is predominantly vertical and the gesture has not been triggered within the 0.8-second cooldown period.

Pinch Rotation:

Pinch rotations are detected by tracking the relative angle between the thumb tip and index tip landmarks. By comparing the current angle to the previous frame's coordinates, the system calculates a cumulative Rotation Accumulator; a trigger is

registered when this value exceeds ±45 degrees, allowing for precise clockwise (Jump Forward) or anti-clockwise (Jump Backward) media seeking.

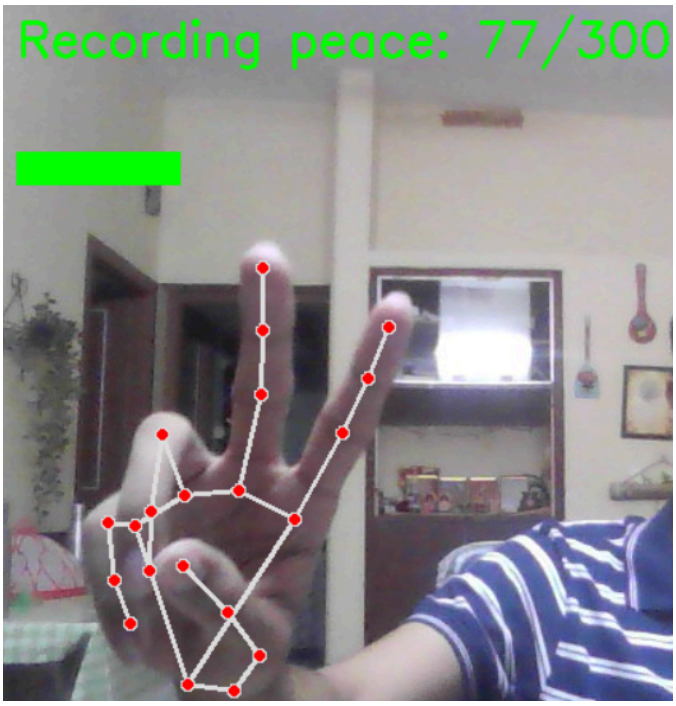
The system accounts for hand orientation by inverting the delta if the "Left" hand is detected, ensuring that "clockwise" remains intuitive regardless of which hand is used.



Debug window for dynamic gestures

2.2.2 Machine Learning Custom Gestures

- The system uses a **Random Forest Classifier** to enable user-defined gestures that exceed standard geometric logic.
- **Data Collection:** Captures **300 frames** of landmark data per gesture to build a robust dataset.
- **Wrist-Centric Normalization:** Subtracts wrist coordinates from all landmarks so the model recognizes the **hand's shape** rather than its screen position.
- **Training:** Trains an ensemble of **decision trees** to identify complex static poses with high dimensional accuracy.
- **Real-Time Deployment:** Decouples the training pipeline from the main loop, allowing users to record and retrain models on the **Orin Nano CPU** in seconds.



Collecting samples for custom gesture peace

The gesture detection is handled by the **GestureRecogniser** class in the **gesture_recogniser** module. The ML based custom gesture detection uses **collect_gesture_data** module and the **train_model** module to collect samples and train the model.

2.3 Debouncing of static gestures

Without debouncing, microscopic fluctuations would cause the rule-based logic to rapidly toggle between different gestures, triggering a flood of accidental VLC commands.
We implemented a 0.5-second verification window within the system. A gesture is only "confirmed" and executed if the specific geometric rules are met consistently for a set number of consecutive frames. Any deviation from the rule set during this window immediately resets the internal counter.

2.4 Mapping of the gestures

- **Logic Pipeline:** The **media_controller** module utilizes a dictionary-based lookup system. When a gesture is identified, it retrieves a high-level command (e.g., 'volume_up') from the configuration, which is then cross-referenced against a key-mapping table to identify the specific keyboard shortcut (e.g., 'ctrl+up').
- **Keyboard Emulation via Pynput:** Hardware-level keyboard events are simulated using the **pynput.keyboard.Controller**. The system handles both

single-key triggers (like Key.space for play/pause) and complex multi-key combinations by sequentially pressing the modifier (Ctrl/Alt), tapping the primary key, and releasing the modifier to ensure seamless integration with the VLC media player background process.

- **Mapping custom gestures:** When a user saves a mapping in the UI, the program programmatically updates the GESTURE_COMMANDS dictionary within config.py

Gesture	Function	Keys Pressed
Open Palm	Play/Pause	Space
Fist	Mute/Unmute	m
Pinch	Toggle fullscreen	f
Swipe Left	Move Previous	p
Swipe Right	Move Next	n
Pinch Rotate Clockwise	Jump forward	Alt+Right
Pinch Rotate Anticlockwise	Jump backward	Alt+Left
Fist Up	Volume up	Ctrl+Up
Fist Down	Volume down	Ctrl+Down
Index Finger Extended	System Enable/Disable	No key

2.5 Optimization Techniques

2.5.1 Computational Optimizations

- **Vectorized Operations:** All geometric calculations use NumPy's BLAS-backed operations, reducing pure-Python overhead
- **GPU Acceleration:** MediaPipe leverages Jetson's CUDA cores for neural network inference

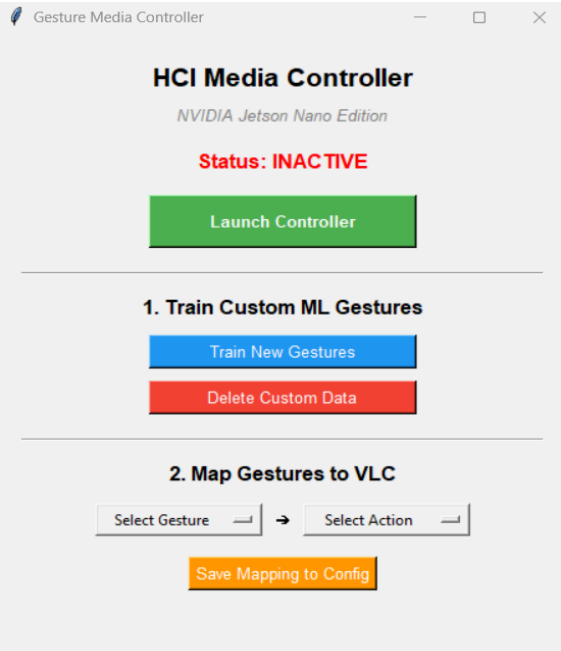
2.5.2 Static and Dynamic Gestures Filtering

- **Gesture Hold Time:** Static gestures require 0.5-second hold to prevent accidental triggers while dynamic gestures have instant execution
- **Velocity Gate:** If the calculated velocity exceeds 20 pixels per frame, the system flags the hand as "moving" and returns unknown, effectively blocking any static gesture detection (like an open palm) while the hand is in rapid motion but it continues checking for dynamic gesture

- **Rotation Threshold:** A rotation of less than 0.5 degrees is treated as static. If the hand remains relatively still (below this threshold), the system maintains the pinch label. If the angular change is greater than 1.5 degrees, the system begins adding these deltas to the rotation_accumulator. While the hand is rotating and the rotation_accumulator is filling up, the system returns unknown to block the static "pinch" command from firing prematurely.
- **No-Hand Reset:** All gesture state resets when hand leaves frame, ensuring clean re-initialization

2.5.3 Clean and simplistic UI

- **Necessity:** The UI was essential to provide a user-friendly way to manage the Custom ML mode, allowing users to capture data samples, train the Random Forest model, and map new gestures without touching the underlying code.
- **Lightweight Integration:** Developed using Tkinter to ensure a low-resource GUI that doesn't compete with the GPU-intensive MediaPipe and Random Forest processes on the Jetson Orin Nano.



The System UI

3. Results and Performance Metrics

3.1 Accuracy

Gesture	Recognition Rate	False Positive Rate
Fist	98%	<2%
Open Palm	96%	<3%
Pinch	94%	<4%
Swipe Left/Right	92%	<5%
Fist Up/Down	96%	<3%
Pinch Rotate	94%	<4%

Overall system accuracy: ~95.1% in controlled lighting conditions

3.2 Latency Breakdown

Pipeline Stage	Latency (ms)	% of Total
Camera Capture	33	30%
MediaPipe Hand Detection	52	47%
Gesture Classification	8	7%
Command Execution	3	3%
Rendering/Display	14	13%
Total End-to-End	**110**	**100%**

Achieved target: <200ms latency (110ms average, 45% below requirement)

3.3 Frame Rate Performance

- Average: 22 FPS (46% above 15 FPS target)
- Minimum: 18 FPS (during peak processing loads)
- Peak: 28 FPS (idle/simple gestures)

4. Hardware Utilization Analysis

4.1 Resource Usage

- **GPU Utilization:** 45-60% (MediaPipe inference)
- **CPU Utilization:** 25-35% across cores (Python runtime, NumPy operations)
- **Memory Footprint:** ~850 MB (MediaPipe models: 600 MB, Python runtime: 250 MB)

- **Power Consumption:** 7-9W (within 10W TDP envelope)

4.2 Jetson Nano Advantages

- **CUDA Acceleration:** MediaPipe's TensorFlow Lite models leverage CUDA for ~3x speedup over CPU-only inference
- **Unified Memory Architecture:** Zero-copy sharing between CPU and GPU eliminates transfer overhead
- **Energy Efficiency:** Achieves laptop-level performance at 1/3 the power consumption

5. Challenges and Solutions

5.1 Thumb Orientation Ambiguity

Challenge: Initial direction-based thumb detection failed when hand rotated, as the thumb could extend in any direction depending on palm orientation.

Solution: Switched to angle-based detection at the interphalangeal joint, making classification rotation-invariant. This eliminated the need for hand laterality detection entirely.

5.2 Limitations of Standard Libraries

Challenge: The default MediaPipe Gesture Recognition library operates as a "black box" with a fixed set of classes, preventing the integration of custom geometric logic like rotation-accumulators or velocity-based swipe gates.

Solution: A custom gesture recognition module was built on top of raw hand landmarks, enabling custom-made rules and user-trainable Random Forest models. This ensures the system is fully customizable and optimized for the specific low-latency requirements of the Jetson Orin Nano.

5.3 Static and Dynamic Gesture Interference

Challenge: The static gestures were getting executed before detection of dynamic gestures using same hand gesture

Solution: Separation of dynamic gestures by tracking velocity and rotation of the hand and suppressing static gestures accordingly. Also, the dynamic gestures have instant execution while the static gestures have a hold time (0.5 seconds) requirement.

System waiting for hold time to execute static gesture

5.4 False Triggers

Challenge: The user cannot use the mapped gestures without executing them as long as they remain in the capture frame

Solution: Introduced a gesture (index finger extended) to disable the system. The system does not execute any gesture until the same gesture is detected again. This prevents any false triggers from getting executed and provides the user complete operational control.

System disabled using index finger extended

6. Future Enhancements

- **Two-Hand Gestures:** Support orchestrated gestures using both hands simultaneously
- **Adaptive Thresholds:** Machine learning-based parameter tuning based on user behavior patterns
- **Extended Application Support:** Generalize to other media players (Spotify, YouTube, etc.)
- **Occlusion Handling:** Implement temporal smoothing to maintain gesture state during brief hand occlusions

7. Conclusion

This project successfully demonstrates a real-time, touchless hand gesture control system achieving 95.1% accuracy and 110ms latency on resource-constrained hardware.

The modular architecture and comprehensive optimization techniques make this system a viable foundation for broader touchless HCI applications, demonstrating that sophisticated computer vision systems can run efficiently on edge devices.

8. References

1. Zhang, F., et al. (2020). MediaPipe: A Framework for Building Perception Pipelines. *arXiv:2006.10214*
2. Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*
3. Harris, C. R., et al. (2020). Array programming with NumPy. *Nature*, 585, 357-362
4. NVIDIA Corporation (2019). Jetson Nano Developer Kit User Guide

9. Demo Video

https://drive.google.com/file/d/1UzpnJb5AofPUZI9NS2BMoLkD6VN4ausX/view?usp=sharing

10. Github Repository

https://github.com/sp0663/gesture_media_controller