# Pose-to-Wave: A Real-Time Gesture-to-Sound Synthesizer Using Arm and Hand Pose Estimation

Luke Qiao and Hannah Shu
Stanford University
lkqiao@stanford.edu, hshu100@stanford.edu

***Abstract*—This work presents Pose-to-Wave, a real-time audio synthesis system that uses machine learning (ML) models to transform multi-person arm and hand movements into expressive digital sound through a combination of modern pose-estimation techniques and classical digital signal processing (DSP) techniques. The system integrates YOLOv8-Pose for six-joint upper-limb tracking and MediaPipe Hands for detailed five-fingertip tracking for each user. These continuous waveforms form the basis functions of an eight-harmonic additive synthesizer. In addition to the core pose-to-audio pipeline, we implement a lowpass filter (LPF) and highpass filter (HPF) to simulate an equalizer (EQ), a flanger effect, and pedal-like echo. The real-time audio engine supports recording, enabling users to capture performances directly from the synthesized output buffer. We successfully implemented these additional features to create a more complete digital musical instrument that connects DSP to the human visceral experience, allowing the user to feel the relationship between motion and sound. As such, Pose-to-Wave opens up new possibilities for artistic expression, providing a controllable and interpretable mapping between human motion and sound.**

***Index Terms*—Digital Signal Processing, Additive Harmonic Synthesis, Computer Vision, Machine Learning, Pose Estimation, Human–Computer Interaction, Real-Time Audio.**

## I. Motivation and Background

The connection between physical movement and sound has long been explored in both artistic practice and technical research. Early gesture-controlled musical systems often relied on specialized hardware such as instrumented gloves, depth sensors, or custom motion-tracking devices to translate human motion into audio. However, advances in computer vision based pose estimation ML models enable expressive, real-time motion-to-sound interaction using only a standard webcam.

Pose-to-Wave synthesizer leverages this capability to map human arm and hand poses directly to synthesized waveforms. By tracking key 2D joint positions such as the user's fingertips, shoulders, elbows, and wrists, the system interprets their geometric configuration as a waveform shape: smooth poses produce sine waves, right-angle bends produce square waves, diagonal ramps produce sawtooth waves, and symmetric "V" shapes produce triangle waves. These detected waveforms are synthesized into audio in real time, allowing physical gestures to function as a dynamic control surface for the synthesizer. The synthesizer also supports multiple users at once, allowing for ensemble performances.

A major motivation for this work is its potential for creative expression. Treating pose as an instrument invites new forms of performance that blend gesture, dance, and digital synthesis. This opens a space for exploration and artistic discovery, demonstrating how computer vision driven interfaces can expand the repertoire of real-time digital performance.

This project is also motivated by its pedagogical value. Instead of encountering DSP purely through symbolic representations, users can gain a physical intuition for how waveform geometry shapes sound, such as how curvature smooths harmonics, how sharper transitions emphasize higher-frequency content, and how symmetry influences timbre. The system therefore serves as a bridge between theoretical understanding and experiential learning, providing a direct and immediate way to internalize the behavior of fundamental signal shapes.

In this way, Pose-to-Wave makes DSP concepts more accessible, intuitive, and engaging by linking them directly to the human body. Fundamental topics are often introduced through equations or code, which can feel removed from their perceptual meaning. By letting users generate and manipulate waveforms through their own movement, the system transforms these abstractions into something that can be felt, heard, and explored interactively.

## II. Overview

An overview of the high-level system architecture of Pose-to-Wave is shown below in Fig 1. Computer vision models detect pose landmarks, which are used to construct a waveform (later defined as the PBW). Then, DSP techniques, like LPF and HPF and user-selected audio effects, are applied on this waveform to synthesize a rich variety of complex sounds.

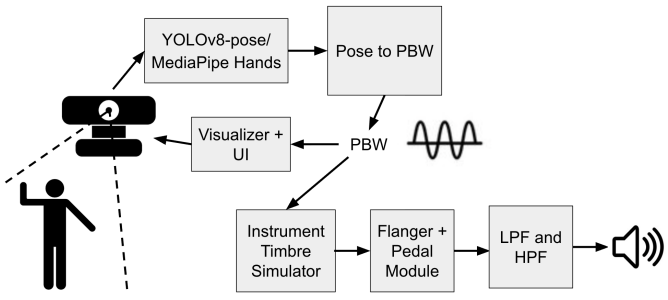


Fig. 1. High-level system pipeline used by Pose-to-Wave.

Pose-to-Wave operates in two primary modes: *Arm Mode* and *Hand Mode*. We design each mode to capture different

types of landmark points. Arm Mode uses the YOLOv8-Pose model to track shoulder, elbow, and wrist landmarks. Hand Mode uses the MediaPipe Hands model to extract the five fingertip positions. Note that Hand Mode provides finer spatial control and higher dexterity, whereas Arm Mode is more optimal for larger gestures. Each arm or hand pose that the models detect in the camera feed introduces a new user with independent controls, allowing for multiple people to produce sounds on the synthesizer at once.

### *2.1 DSP Features*

First, we convert the pose geometry into a waveform defined over a single period. Because the number of detected landmarks is limited, we upsample this single-period waveform to increase its resolution and obtain finer control over its shape. We then construct its periodic extension, which we refer to as the pose-based waveform (PBW). The PBW serves as the basis for the system's synthesis pipeline, and is ultimately processed and outputted as audio.

The core DSP pipeline centers on frequency-domain processing and additive harmonic synthesis. After converting the pose to a PBW, the system constructs its frequency spectrum using the Discrete Fourier Transform (DFT), implemented as the Fast Fourier Transform (FFT), enabling fast analysis and manipulation of the harmonic content generated by the user's pose, which is important for achieving real-time speeds.

The frequency spectrum of the PBW is used for features like additive harmonic synthesis to reconstruct more complex instrument-like timbres. We use the PBW as a basis function from which an overtone series of eight harmonics is generated. Each integer multiple of the fundamental frequency is weighted according to a selected harmonic profile. These profiles are inspired by real acoustic instruments like the piano and violin, and allow the synthesized sound to take on distinct timbral identities.

To enhance the sonic flexibility for the synthesizer, we also implement several standard DSP effects:

- A LPF and HPF to simulate EQ-like control of the PBW.
- A flanger effect to simulate the repeated plucking of a note to create variety in texture.
- A pedal-like echo that adds spatial depth and lingering decay to gestures.

These processing blocks operate in real time and can be combined to produce a diverse range of timbres and textures. However, to remove microtones that cause unpleasant intervals, we also implement frequency rounding to the nearest note in a user-defined scale, such as C major or F# minor. This allows the synthesizer to be able to generate more desirable tonal melodies and harmonies.

### *2.2 System Overview*

Pose-to-Wave runs as a continuous real-time loop operating at typical webcam frame rates of 20-30 FPS. Each iteration captures an image, performs pose detection, constructs the PBW, synthesizes audio at sampling frequency $f_s = 44.1\,\text{kHz}$, and updates the visualization interface. The system's architecture pipelines video capture, ML model inference, waveform generation, audio synthesis, and plotting to minimize blocking operations. Because inference latency can fluctuate, the audio engine is decoupled and buffered to prevent dropouts and maintain stable musical feedback.

A live visualization displays the pose overlay, audio effect settings, and the corresponding waveform and frequency spectrum, allowing users to simultaneously see and hear the influence of their movements on the synthesized sound.

## III. Implementation

### *3.1 Computing the PBW*

Computing the PBW from the pose is the core of the synthesizer. More formally, we construct the map $\mathcal{M} : \mathcal{P} \to \text{PBW}$, where $\mathcal{P}$ is the set of all pose configurations. The system first normalizes the detected landmark coordinates to remove scale and translation effects. Because the raw pose landmarks are sparse, as mentioned in the overview, we upsample the single-period waveform $x_p[n]$ to obtain a higher-resolution sequence that provides finer control over the resulting shape. To approximate the infinite time-domain sinc interpolation (ideal brick-wall LPF), we upsample by applying a cubic interpolation to form a smooth, continuous curve.

To produce a periodic waveform suitable for audio synthesis, we then resample $x_p[n]$ to a length $N$ that corresponds to the desired pitch, where $N \approx f_s/f$ and $f_s = 44.1$ kHz is the audio sampling rate and $f$ is the pose-derived fundamental frequency. The pose-based waveform is then defined as the modulo-$N$ periodic extension

$$\text{PBW} := x_p[n \bmod N].$$

Note that it suffices to store the single-period waveform $x_p[n]$ instead of the theoretically infinite-time PBW. This construction allows Pose-to-Wave to synthesize the user-defined waveform directly rather than relying on fixed lookup tables, enabling a continuous spectrum of intermediate and hybrid shapes beyond standard analytic waveforms (e.g., sine, square, sawtooth, triangle).

Notice that we must handle the inevitable case of "invalid" poses, as shown below in Fig 2. An invalid pose is any pose

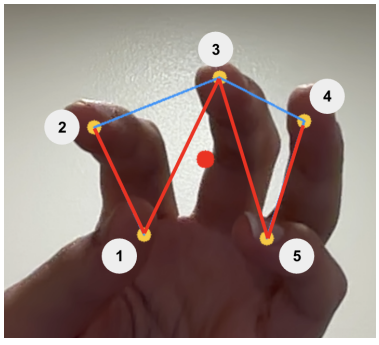


Fig. 2. Example of an invalid pose configuration.

that would cause self-overlap in the PBW. In this case, if the

raw waveform were periodically extended, the PBW would exhibit self-overlap because point 1 lies to the right of point 2 and point 5 lies to the left of point 4. Then, we see that the points and connecting lines that are fed in from the pose estimation model must be rewired such that their x-coordinates are sorted in increasing order, as shown by the red line. Using this method, the first and last points per period are guaranteed to form a valid PBW without self-overlap when connected. Therefore, all pose coordinates from the camera feed are first sorted by x-coordinate before further processing to ensure that all resulting signals are valid.

### *3.2 Spacial Feature Metadata*

Additional spatial features, such as pose width or average landmark position, are encoded as metadata and used to control properties of the PBW.

*3.2.1 Frequency Mapping:* In the previous section, it was discussed that the fundamental frequency $f$ of the PBW is derived from the pose. Indeed, the pose width directly determines $f$ and therefore indirectly sets the PBW length $N$. After sorting and normalization, the system computes the cumulative horizontal arc-length of the pose landmarks. Let $x_p[n]$ denote one period of the PBW. We form the incremental distances

$$d[n] = \|x[n] - x[n-1]\|_2^2,$$

and compute the cumulative sum

$$t[n] = \sum_{k=1}^{n} d[k], \;\; t[0] = 0.$$

The total width of the pose is then $\texttt{raw\_width} = t[N-1]$, which increases when the user stretches their pose farther apart or produces a horizontally long waveform shape.

To map this geometric spatial width into a normalized control variable, we define fixed bounds $\texttt{RAW}_{\min}$ and $\texttt{RAW}_{\max}$ depending on the camera feed window size and compute

$$\texttt{norm} = \frac{\texttt{RAW}_{\max} - \texttt{raw\_width}}{\texttt{RAW}_{\max} - \texttt{RAW}_{\min}}, \;\; \texttt{norm} \in [0,1].$$

This choice ensures that narrower poses correspond to values of norm near 1 and wider poses correspond to values near 0, producing the intuitive behavior of higher pitches for narrower poses.

To decrease sensitivity and produce smoother transitions, we use a nonlinear shaping exponent $\gamma = 0.4$ to define the parameter $v = \texttt{norm}^{\gamma}$. The value $v$ is then converted into the fundamental frequency of the PBW using an exponential interpolation between a minimum and maximum playable pitch $f_{\min}$ and $f_{\max}$, given by

$$f := f_{\min} \left( \frac{f_{\max}}{f_{\min}} \right)^{v}, \;\; f \in [f_{\min}, f_{\max}].$$

Exponential interpolation is used because musical pitch is perceived logarithmically, so the mapping above produces musically natural pitch changes as the user varies the pose width. We set $f_{\min} = 65.41$ Hz and $f_{\max} = 1046.5$ Hz, corresponding to a four octave pitch range from C2 to C6.

Finally, the frequency is quantized to the nearest pitch in the user-selected musical scale is

$$f_q = \texttt{QuantizeToScale}(f, \text{scale}),$$

ensuring that the generated audio remains harmonically consistent with the selected key.

*3.2.2 LPF and HPF Mapping:* The metadata also contain the normalized average position of all detected landmarks $(\bar{x}, \bar{y})$. This metadata is used to control the LPF and HPF that is continuously applied on the synthesized PBW, further allowing the user to shape the synthesized spectrum with EQ-like controls. The average vertical position $\bar{y}$ controls the LPF cutoff frequency $f_{\text{LPF}}$ and the average horizontal position $\bar{x}$ controls the HPF cutoff frequency $f_{\text{HPF}}$, forming an intuitive 2D mapping between average location and timbre.

In order to reduce jitter in the filter cutoff frequencies from small variations in position, each normalized coordinate is discretized into one of eight bins

$$b_x = \lfloor 8\bar{x} \rfloor, \;\; b_y = \lfloor 8\bar{y} \rfloor, \;\; b_x, b_y \in \{0, \ldots, 7\}.$$

These bin indices select cutoff frequencies from precomputed linearly spaced ranges

$$f_{\text{LPF}} := \texttt{LPF\_BINS}[b_y], \;\; f_{\text{HPF}} := \texttt{HPF\_BINS}[b_x],$$

where $\texttt{LPF\_BINS}$ spans $[555.6\,\text{Hz}, 4000\,\text{Hz}]$ and $\texttt{HPF\_BINS}$ spans $[10\,\text{Hz}, 555.6\,\text{Hz}]$. Notice that $555.6$ Hz was chosen to be the midpoint of the frequency range of the synthesizer.

Note that the filters are implemented as 4th order Butterworth filters in second-order-section (SOS) form, a numerically stable representation that decomposes the filter into two cascaded 2nd order biquads rather than using a single 4th order difference equation. They are applied in a streaming manner within the audio callback in real time. This means that each audio callback supplies a short block of 512 samples, and the Butterworth IIR filters are applied to this block using `sosfilt`, which internally evaluates the corresponding biquad difference equations. Because IIR filtering depends on past inputs and outputs, each filter maintains a small set of internal "state" values. These state vectors are updated after each audio block and carried into the next one, ensuring that the filter's recursive dynamics continue smoothly across block boundaries without producing clicks or discontinuities. Therefore, this effectively executes the recursive update

$$y[n] = \sum_k b_k x[n-k] - \sum_k a_k y[n-k],$$

while ensuring that the required history terms persist between blocks. This streaming architecture prevents clicks or discontinuities and enables stable real-time spectral shaping of the synthesized sound based on the user's pose.

Raising the pose reduces the LPF cutoff, darkening the sound, while moving the pose horizontally increases the HPF cutoff, brightening the sound. Together, these mappings provide stable, real-time, pose-driven control over the spectral envelope of the synthesized audio.

### 3.3 User Controlled Effects

In this section, we discuss the implementation of three user controlled audio effects that are applied on the PBW: 1) an instrument timbre simulator, 2) a flanger effect, and 3) a pedal-like echo effect.

#### 3.3.1 Instrument Timbre Simulator:

*Instrument Timbre Modeling:* We implement instrument-inspired timbres to allow for the user to customize the complexity of the synthesized sound beyond filtering. We do this with an eight-harmonic weighting model, using the PBW as the basis function. Each instrument is represented by a harmonic amplitude vector

$$\mathbf{w} = [w_1, w_2, \ldots, w_8],$$

which specifies the relative strength of the first eight partials. These profiles encode characteristic spectral envelopes (e.g., the piano's strong low-order harmonics, the flute's rapidly decaying upper partials, the clarinet's odd-harmonic emphasis, or the violin's rich upper-harmonic content).

Let $x_p[n]$ denote the single-period PBW of length $N$. To synthesize a specific timbre, the system constructs each harmonic by time-compressing the PBW by an integer factor. More precisely, the $k$-th harmonic is generated by

$$x_p^{(k)}[n] = x_p[(kn) \bmod L],$$

which produces a waveform with the same shape but $k$ times the fundamental frequency. The final instrument waveform is obtained by an amplitude-weighted sum

$$x_{p,\text{instr}}[n] := \sum_{k=1}^{8} w_k x_p^{(k)}[n]$$

This method differs from classical additive synthesis (which uses pure sinusoids) by using the PBW itself as the basis function for all harmonics. As a result, the synthesized timbre simultaneously reflects the selected instrument profile and preserves movement-dependent nuances of the user's pose. Because each harmonic is derived from the PBW rather than an analytic basis, the system naturally produces expressive, hybrid timbres that are inspired by acoustic instruments, but produce unique sounds.

#### 3.3.2 Flanger Effect:

A flanger is a classic audio effect that creates a characteristic "whooshing" or "jet-plane" sound by mixing a signal with a slightly delayed and continuously modulated version of itself. This modulation introduces a shifting comb-filter pattern in the frequency domain, producing sweeping notches that move over time.

We implements the flanger effect as a real-time, modulated delay-line structure. The effect is produced by summing the dry signal with a time-varying delayed copy:

$$y[n] = x[n] + \alpha\, x[n - d(n)],$$

where $x[n]$ is the input sample, $\alpha = 0.5$ is the wet-mix coefficient, and $d(n)$ is a delay measured in samples. The delay is modulated by a sinusoidal low-frequency oscillator (LFO), which smoothly varies the delay between a base value of 3 ms and an upper limit of 5 ms:

$$d(n) = d_{\text{base}} + d_{\text{depth}}\Big(\frac{\sin\theta(n) + 1}{2}\Big),$$

where $\theta(n)$ is the LFO phase updated each sample according to the user-specified modulation rate.

A circular delay buffer stores recent audio samples, enabling efficient access to $x[n - d(n)]$. The buffer indices and LFO phase are carried across audio callback blocks, ensuring continuous modulation without discontinuities. After mixing, a normalization step prevents clipping. This streaming implementation allows the flanger to operate stably in real time.

#### 3.3.3 Pedal-Like Echo:

The synthesizer includes a pedal mode that emulates a piano-style sustain by adding an exponentially decaying echo tail to the signal, even after new input has stopped. We implement this effect using a short feedback–comb structure: a delay buffer of length $D = \texttt{REVERB\_DELAY\_SEC} \cdot f_s$ stores the most recent audio, and each output sample is computed as

$$y[n] = x[n] + b[n],$$

where $b[n]$ is the delayed content taken from the circular buffer. The buffer is updated according to

$$b[n] = x[n] + \alpha b[n - D],$$

with feedback gain $\alpha$ chosen so that the echo decays to roughly $1\%$ after the user-selected sustain time $T = \texttt{PEDAL\_TIME}$. This is computed in the code via

$$\alpha = 0.01^{D/T}.$$

When pedal mode is enabled, the buffer continuously recirculates its own past energy, allowing the sound to "ring" for several seconds. When pedal mode is disabled, the buffer is hard-cleared to prevent leftover tail artifacts. The algorithm operates entirely in the audio callback, enabling real-time sustain behavior without interfering with pose tracking or synthesis.

### 3.4 Multi-Threaded Audio Engine

Pose-to-Wave generates audio through a callback-based real-time synthesis engine built on the `sounddevice` (PortAudio) framework. Unlike the pose-detection and terminal-control threads, which operate asynchronously and at variable frame rates, audio synthesis must run under strict real-time deadlines. We therefore separate audio generation into a dedicated high-priority callback that is invoked periodically by the audio driver.

#### 3.4.1 Audio Callback:

The audio engine is initialized by creating an `OutputStream` with a sampling rate of $f_s = 44.1$ kHz and a block size of 512 samples. When the stream is started, PortAudio repeatedly invokes a user-defined function `audio_callback()`, which must fill the output buffer `outdata` with exactly 512 samples before the next deadline.

Within this callback, the system reconstructs audio by reading through the single-period waveform. Each detected user contributes one "voice," where we have that the $v$-th voice is represented by:

- a single-period waveform $x_{p,v}[n]$,
- a phase accumulator $\phi_v$ indicating the current index inside the wavetable,
- LPF and HPF states $(\mathbf{z}_v^{\text{LP}}, \mathbf{z}_v^{\text{HP}})$ for continuity across adjacent blocks.

The callback computes the next 512 samples of the PBW of each voice $y_v[n]$ by indexing into $x_{p,v}[n]$ with a running phase, given by

$$y_v[n] = x_{p,v}[(\phi_v + n) \bmod N_v],$$

where $N_v$ is the period length in samples. After processing all voices, the callback updates each $\phi_v$ so that the next block continues exactly where the previous one ended, ensuring seamless periodic playback.

*3.4.2 Shared State Protection:* The audio callback runs in a real-time thread and must access several shared DSP data structures, including the list of pose-based waveforms, per-voice phase accumulators, Butterworth filter coefficients, and IIR filter memories. These structures are simultaneously modified by the pose-detection thread and the terminal-command thread, creating the potential for race conditions. For example, if the audio callback reads a filter memory vector while another thread is resizing or replacing it, the result may be corrupted state, inconsistent list lengths, or audible artifacts.

To ensure correctness, we protect all shared DSP structures using a `threading.Lock`. The lock defines a critical region in which only one thread may read or write shared state at a time. The audio engine uses a copy-compute-writeback pattern designed to avoid blocking the real-time thread:

- At the beginning of each callback, the audio thread acquires the lock and copies the current periods, phase accumulators, and IIR filter memories into private local variables. This operation is extremely fast and the lock is released immediately afterward.
- All expensive operations, like filtering and pedal/flanger audio effects, are performed without holding the lock. This allows the pose-detection and command threads to update shared state freely during audio processing.
- Once a new audio block has been synthesized, the audio thread reacquires the lock and writes back only the updated filter memories and phase accumulators. This ensures continuity across blocks while keeping lock usage minimal.

This design prevents race conditions, guarantees glitch-free real-time audio, and allows the vision thread to update voice parameters at video-frame rates (20-30 Hz) without interfering with audio performance.

*3.4.3 Independence from External Latency:* A critical architectural decision is that the audio callback never invokes the camera, pose-estimation model, or any slow computation that introduces external latency. Indeed, all vision processing occurs in a separate thread:

- The pose-detection thread periodically computes new PBWs, fundamental frequencies, filter cutoffs, and instrument assignments.
- These values are published to the audio engine via the function `update_audio_from_multiple()`, which safely replaces the shared DSP state under lock protection.
- The callback then renders audio using the most recently provided state snapshot, without the need to wait on pose detection.

Because the callback always produces audio based on the latest available state, but never blocks while waiting, the audio output remains stable even if video processing experiences temporary delays.

Therefore, the architecture we implement guarantees that every audio block is generated within a bounded time budget and is unaffected by I/O delays. At the same time, pose updates are incorporated with a latency of one audio block (512 samples $\approx$ 11.6 ms), enabling responsive and stable multi-voice synthesis.

### *3.5 User Interface and Controls*

Finally, we discuss the user interface and controls that the users interact with to control settings like the instrument timbre, flanger effect, and pedal effect. The user interface also features visualizations of the single-period waveform and overlays on the camera feed showing the detected pose landmarks and LPF and HPF cutoff frequencies.

*3.5.1 Custom Terminal Command Interface:* We created custom terminal commands to control the synthesizer. This runs in a dedicated background thread `terminal_command_loop()`. When the system starts, this thread prints an interactive help menu listing all available commands for controlling the synthesizer. The supported actions include:

- **Mode selection:** `mode hand` or `mode arm` switches between MediaPipe fingertip tracking and YOLO-based arm pose estimation.
- **Musical scale selection:** `scale <name>` (e.g., "C major", "Eb minor") updates the global scale used to quantize pose-derived frequencies to musical pitches.
- **Instrument assignment:** `instrument personN <name>` assigns one of the timbre profiles (piano, flute, violin, *etc.*) to a specific detected user. Note that this assignment is persistent.
- **Effects processing:** These settings change the audio effects applied to the PBW.
  - `pedal on/off` and `pedal time <sec>` to control the sustain/echo effect,
  - `flanger on/off`, `flanger rate <Hz>`, and `flanger depth <ms>` to shape the modulation effect.

- **Recording:** `record start <name>`, `record stop`, and `record status` allow users to capture audio to WAV files, saved to a file named `<name>.wav`.

Each command is parsed and dispatched to the corresponding setter function in `freq_processing.py` (e.g., `set_pedal_mode()`, `set_flanger_params()`, `set_person_instrument()`). These setters update the shared DSP state with lock protection so that the audio callback can safely use the new parameters.

*3.5.2 Visualizations:* Real-time visual feedback is rendered directly onto the camera image stream. For each incoming frame, we show the detected hand or arm landmarks and their connecting segments, displays per-person musical metadata, and prints a mode indicator in the top-left region of the frame. These visual overlays provide immediate feedback about pose geometry, filter settings, and musical quantization.

*3.5.3 Waveform and FFT Visualization:* Beyond the camera overlays, Pose-to-Wave includes a signal analysis visualizer. When the system initializes, `init_plot()` opens an interactive Matplotlib window, displaying the single-period waveform after instrument shaping and the magnitude FFT in dB.

The function `update_plot()` is called once per video frame to refresh all waveform and FFT curves. Because these plots show precisely the signal passed into the real-time synthesizer, users can see the effects of harmonic profiles, LPF/HPF shaping, pose changes, and audio effects in real time.

## IV. Results

### *4.1 Harmonic Structure Verification Across Instrument Modes*

To evaluate whether the implemented harmonic profiles produce their intended spectral signatures, we generated spectrograms for four different instrument modes: the fundamental mode, piano, clarinet, and violin. Each mode corresponds to a predefined amplitude vector specifying the relative strength of the first eight harmonics. These amplitude profiles are:

$$\text{fundamental} = [1.0, 0, 0, 0, 0, 0, 0, 0]$$

$$\text{piano} = [1.0, 0.75, 0.55, 0.40, 0.28, 0.20, 0.14, 0.10]$$

$$\text{clarinet} = [1.0, 0.0, 0.7, 0.0, 0.5, 0.0, 0.3, 0.0]$$

$$\text{violin} = [1.0, 0.95, 1.0, 0.95, 0.85, 0.75, 0.65, 0.55]$$

The spectrograms generated from the synthesized output exhibit clear harmonic bands whose relative intensities align with these amplitude patterns.

Note that the spectrograms were recorded across different hand poses and a range of user-generated frequencies. Because real-time gesture control naturally changes pitch and timbre, comparing different instrument modes directly is not meaningful. Instead, our evaluation focuses on verifying that each instrument mode consistently matches its own harmonic profile, regardless of the specific pose or frequency used. This confirms that the implemented amplitude vectors reliably generate the expected timbral identity under natural user variation.

*4.1.1 Fundamental Mode:* The fundamental instrument mode spectrogram (Fig 3) shows a single dominant harmonic at the base frequency with negligible energy at higher multiples. The harmonic lines above the first appear at lower amplitudes. This matches the amplitude vector, which suppresses all overtones. The overtones we see on the spectrogram come from the fact that the fundamental itself is not a pure tone.

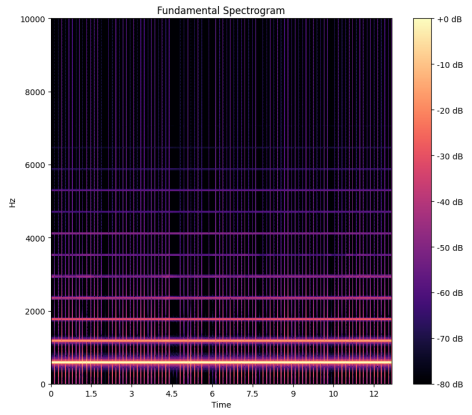


Fig. 3. Fundamental Instrument Mode Spectrogram

*4.1.2 Piano Mode:* The piano instrument mode spectrogram (Fig 4) decays smoothly and monotonically across harmonics, producing a bright but controlled spectrum. In the piano spectrogram, the first few harmonics (up to around 3000-5000 Hz) have strong, clearly defined energy bands. Higher harmonics gradually taper off, but remain visibly present. This correlates well with the amplitude coefficients, as expected.

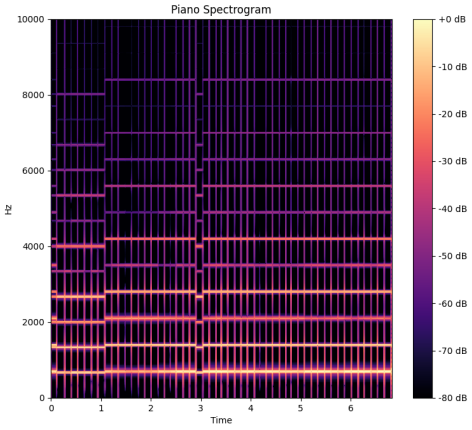


Fig. 4. Piano Instrument Mode Spectrogram

*4.1.3 Clarinet Mode:* The clarinet instrument mode spectrogram (Fig 5) exhibits an alternating pattern of strong and suppressed harmonics, reflecting the instrument's odd-harmonic dominance. The defined amplitudes produce strong

first, third, fifth, and seventh harmonics, with even harmonics nearly absent.

This pattern appears in the clarinet spectrogram, since every pair of harmonic bands exhibit the same amplitude, as expected.

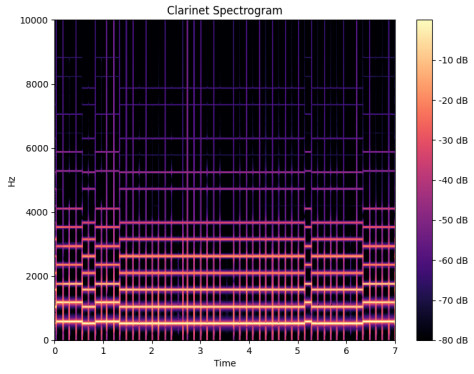


Fig. 5. Clarinet Instrument Mode Spectrogram

*4.1.4 Violin Mode:* The violin instrument mode spectrogram (Fig 6) is intentionally dense, with nearly all harmonics weighted close to 1.0. This produces a rich, full-spectrum sound characteristic of bowed string instruments.

The violin spectrogram confirms this, since the overtones all remain strong in amplitude, with only a gradual decay at higher orders. Unlike the clarinet or piano, the violin displays sustained spectral density. The nearly equal strength of the first four harmonics is particularly visible in the spectrogram, matching the amplitude specification.

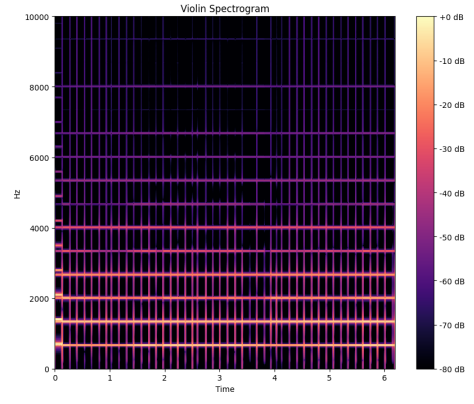


Fig. 6. Violin Instrument Mode Spectrogram

## *4.2 Pose-Waveform Correspondence*

A core goal of Pose-to-Wave is to allow users to intentionally shape sound by intentionally shaping their bodies. To demonstrate this, we conducted a series of qualitative pose tests in which users freely arranged their hands and arms into different geometric configurations. For each pose, the system simultaneously records 1) the 2D pose skeleton overlaid on the video frame and 2) the spline-based waveform generated from the detected keypoints. These examples illustrate how any arbitrary arrangement of fingers or arms, whether curved, angled, symmetric, or irregular, maps directly to a corresponding waveform shape.

*4.2.1 Hand Poses:* Hand mode uses the five fingertip coordinates as spline control points, enabling users to freely shape the resulting waveform by forming any configuration with their fingers. The system interprets whatever geometric structure the user creates, curved, angular, symmetric, or irregular, and converts it directly into a continuous waveform. Below are several examples illustrating how arbitrary hand formations translate into distinct waveform shapes.

*Sine Wave:* The user forms a gesture with the index and ring fingers extended (Fig 7). The fingertip positions create a peak on the left side and a valley on the right side. We see that the resulting curve traces a smooth rise and fall that resembles a sinusoidal cycle, as expected.

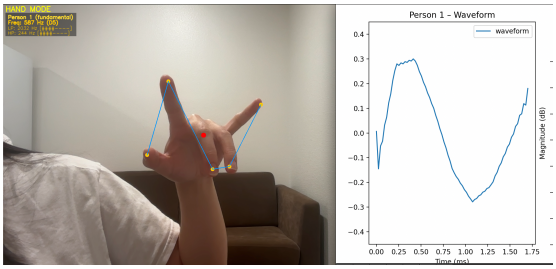


Fig. 7. Sine Wave Hand Gesture

*Sawtooth Wave:* When the user places all fingertips along a descending (or ascending) diagonal, the x-sorted fingertip coordinates form a monotonic decline (Fig 8). The resulting waveform is a smooth ramp function, forming a sawtooth wave.

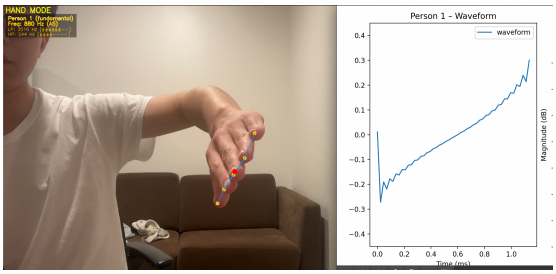


Fig. 8. Sawtooth Wave Hand Gesture

*Rectangular Wave:* When the fingertips curve into a claw shape, the spline control points generate a plateau shaped waveform (Fig 9). The center of the hand forms the highest flat portion of the waveform, while the two outer fingertips form the valleys on both sides. This produces a waveform closely resembling a rectangular wave with finite slew rate.

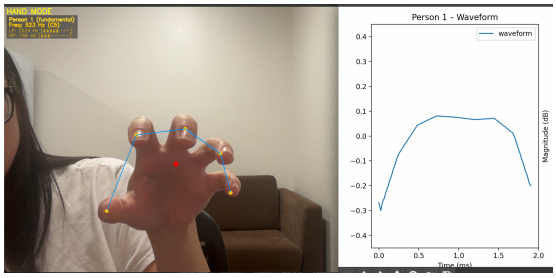


Fig. 9. Rectangular Wave Hand Gesture

*Triangle Wave:* Extending both index and pinky in opposite directions results in a symmetric V-shaped skeleton (Fig 10). The waveform displays an inverted triangle structure, with a sharp minimum corresponding to the centered middle fingertip.

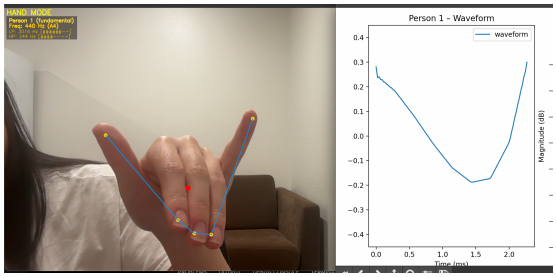


Fig. 10. Triangle Wave Hand Gesture

*4.2.2 Arm Poses:* Arm mode uses YOLOv8-Pose landmarks (shoulders, elbows, and wrists) to extract the overall configuration of the upper limb. Because arm positions often form larger geometric structures, straight lines, angles, arcs, or symmetric arrangements, the resulting splines reflect these broader shapes. Users can move or reposition their arms to produce any desired geometric pattern, and the system maps that pattern directly into the synthesized control waveform.

*Triangle Wave:* The user is pointing both arms diagonally downwards (Fig 11). The spline trace of the six joints yields a waveform with a single smooth peak, closely resembling a triangle wave with a sharp peak.

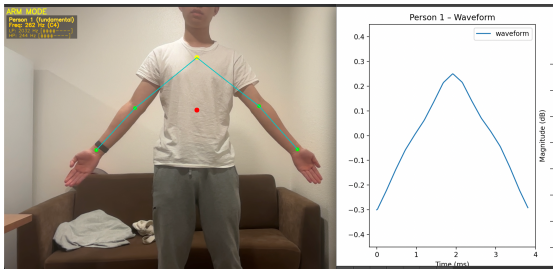


Fig. 11. Triangle Wave Gesture with Arms

*"W" Wave:* We see that bent elbows form a "W" shape, as the waveform exhibits two local minima separated by a central peak (Fig 11). This is consistent with the geometry, since each elbow joint introduces a valley in the spline, while the shoulders create the central peak. The synthesized waveform accurately mirrors the pose structure.

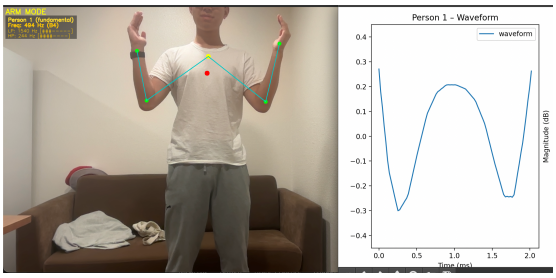


Fig. 12. "W" Wave Gesture with Arms

## V. Conclusions and Future Work

Pose-to-Wave demonstrates that real-time, vision-driven gesture synthesis is not only feasible but capable of producing interpretable audio directly from human movement. By combining modern pose-estimation models (YOLOv8-Pose and MediaPipe Hands) with classical DSP tools, including spline interpolation, frequency-domain analysis, and harmonic additive synthesis, the system establishes a transparent mapping between body geometry and sound. Our experiments show that both small-scale hand gestures and large-scale arm movements reliably generate waveforms whose structure mirrors the underlying pose. The qualitative pose-to-waveform correspondences confirm that the spline-based PBW construction, the width-based frequency mapping, and the instrument-harmonic weighting pipeline operate robustly and intuitively across a wide range of user poses.

Methodologically, several decisions proved essential for stability and expressiveness. The x-sorting of pose landmarks prevented self-overlapping waveforms; cubic interpolation produced smooth, musically meaningful curves; and exponential pitch scaling enabled natural-feeling control over musical intervals. The harmonic modeling results, verified through spectrogram analysis, demonstrated that our eight-partial timbre simulator can accurately approximate the spectral envelopes of instruments such as piano, clarinet, violin, and more. Moreover, the multi-threaded audio architecture ensured that real-time synthesis remained stable despite variability in neural inference latency, enabling glitch-free playback even under visually complex scenes.

### *5.1 Future Work*

Although Pose-to-Wave achieves stable and expressive real-time interaction, several areas offer promising pathways for further development:

1) **Improved Vision Robustness.** In challenging lighting or cluttered backgrounds, YOLOv8-Pose and MediaPipe Hands may occasionally mis-detect extra "ghost" users or jitter the landmark positions, causing unintended pitch fluctuations. Future work includes integrating temporal

smoothing, user identification across frames, or migrating to models optimized for upper-limb stability. Additional post-processing heuristics could also suppress phantom detections and ensure consistent tracking even under occlusion.

2) **Expanded Audio Feature Set.** The current synthesizer includes EQ-style filtering, harmonic modeling, flanger, and pedal-like echo. To broaden expressive capability, future versions will incorporate vibrato, ADSR envelopes, and more. These additions would allow for richer timbral shaping beyond the current effects.
3) **Deployment as a Web Application.** To make the system broadly accessible, future development will port Pose-to-Wave into a browser-based environment. A web implementation would enable anyone with a laptop or smartphone camera to play the instrument instantly, without installations.
4) **More Variation in Timbre** To expand the sonic palette, a future version will introduce a larger library of instruments. Many acoustic instruments, including brass, woodwinds, plucked strings, bowed strings, and synthetic designs, exhibit distinctive spectral envelopes that can be approximated with extended overtone vectors.

## References


[1] A. H. Benade, *Fundamentals of Musical Acoustics*. Oxford University Press, 1976.
[2] M. Bosi, "DFT Implementation (Lecture 10)," Stanford University, 2025.
[3] M. Bosi, "Windowing and Filtering (Lecture 11)," Stanford University, 2025.
[4] M. Bosi, "Lecture 12: Convolution and Filtering," MUSIC 320: Introduction to Digital Audio Signal Processing, Stanford University, 2025.
[5] A. Bouënard, M. M. Wanderley, and S. Gibet, "Virtual gesture control of sound synthesis: Analysis and classification of percussion gestures," *Applied Acoustics*, vol. 96, pp. 668–677, 2015.
[6] L. Cremer, *The Physics of the Violin*. MIT Press, 1984.
[7] W. Fan and X. An, "A deep learning based framework for music-synchronized dance choreography with pose quantization and motion prediction for activity recognition," *Scientific Reports*, vol. 15, Article 37248, 2025. doi: 10.1038/s41598-025-21266-1.
[8] N. H. Fletcher and T. D. Rossing, *The Physics of Musical Instruments*. Springer, 1998.
[9] J.-W. Kim, J.-Y. Choi, E.-J. Ha, and J.-H. Choi, "Human Pose Estimation Using MediaPipe Pose and Optimization Method Based on a Humanoid Model," *Applied Sciences*, vol. 13, no. 4, 2700, 2023. doi: 10.3390/app13042700.
[10] Google Research, "MediaPipe Hands," 2020.
[11] M. Muller, *Fundamentals of Music Processing*. Springer, 2015.
[12] J. O. Smith, *Spectral Audio Signal Processing*. W3K Publishing, 2011.
[13] J. O. Smith, *Spectral Audio Signal Processing*. Stanford University.
[14] Ultralytics, "YOLOv8-Pose Documentation," 2023.


## Appendix: Author Contributions

The Pose-to-Wave system was jointly designed and implemented by Luke Qiao and Hannah Shu. Their individual contributions are listed below.

### *Luke Qiao*

Luke focused on the DSP, synthesis, and systems-engineering components of the project. His primary contributions include:

- Designing and implementing the PBW construction pipeline
- Creating the geometric-to-frequency mapping, nonlinear shaping, and pitch quantization framework.
- Implementing the LPF/HPF real-time filtering architecture using Butterworth filters and metadata mapping.
- Designing the real-time audio engine, including multithreading, buffer synchronization, and callback-based streaming for glitch-free synthesis.
- Designing the user interface and terminal command system for real-time control of modes, effects, and recording.

### *Hannah Shu*

Hannah focused on the DSP, design and implementation of the computer-vision system, and the front-end user interface. Her primary contributions include:

- Integrating YOLOv8-Pose and MediaPipe Hands for multi-user arm and hand tracking.
- Developing the real-time visualizer, including pose overlays, waveform plots, and FFT displays.
- Implementing the instrument timbre system and verifying harmonic structure against expected amplitude profiles.
- Implementing the flanger and pedal-like echo modules, including delay-line structures and LFO modulation.