

Automated Calibration and Semantic Parameter Mapping Frameworks for OSC-Controlled Digital Audio Workstations

1. Introduction: The Semantic Gap in Autonomous Audio Systems

The integration of algorithmic agents with Digital Audio Workstations (DAWs) represents a paradigm shift in music production and audio engineering. By decoupling control logic from human interface constraints, developers can construct systems capable of generative composition, automated mixing, and adaptive sound design. However, this decoupling introduces a fundamental "Semantic Gap" between the internal mathematical representation of audio parameters and the external control protocols used to manipulate them.

In modern audio architectures—exemplified by Ableton Live, Logic Pro, and the VST (Virtual Studio Technology) standard—parameter control is bifurcated into a Model-View-Controller (MVC) pattern. The **Model** operates on high-precision, physical units essential for Digital Signal Processing (DSP): Hertz (Hz) for frequency, Decibels (dB) for amplitude, and milliseconds (ms) for time.¹ The **View**, or the Graphical User Interface (GUI), presents these values to the user in a human-readable format. Crucially, the **Controller** layer—which includes automation lanes, MIDI mapping, and the Open Sound Control (OSC) APIs—abstracts these distinct physical domains into a unified, normalized floating-point coordinate space, typically bounded by the closed interval $[0.0, 1.0]$.³

1.1 The Necessity of Normalization

This normalization strategy is not merely a convenience but an architectural necessity for DAWs that must host third-party plugins. A DAW cannot inherently know that a specific parameter on a synthesizer represents a filter cutoff frequency ranging from 20 Hz to 20 kHz, while another parameter represents a resonance factor from 0.0 to 1.0. To standardized automation recording and hardware surface integration, the host treats all continuous parameters as generic floats. A MIDI fader sending a value of 64 (on a 0-127 scale) is translated to approximately 0.503 , which is passed to the plugin.⁵ The plugin is then solely responsible for "denormalizing" this value into the specific domain required by its DSP engine.

1.2 The Challenge for Algorithmic Agents

For a human operator, this abstraction is transparent. The feedback loop is closed visually: as the user turns a physical knob, the GUI updates to display "1.2 kHz" or "-12 dB," allowing for precise adjustments based on visual confirmation. For a "headless" algorithmic agent operating via OSC, this visual feedback loop is severed. The agent operates in the blind normalized space.

Consider an agent tasked with setting a High-Pass Filter to remove sub-bass frequencies below 120 Hz. If the agent naively assumes a linear mapping over a 20 Hz – 20 kHz range and sends the normalized value corresponding to the arithmetic proportion, the result will be catastrophically incorrect due to the logarithmic nature of frequency parameters. To set the value to 120 Hz, the agent must know the exact **Inverse Transfer Function**

$f^{-1}(120 \text{ Hz}) \rightarrow [0.0, 1.0]$. Since this function varies per parameter—ranging from linear to exponential, logarithmic, or custom piecewise curves defined by "Skew Factors"—the agent cannot reliably control the system without a calibration layer.

1.3 The Auto-Calibration Solution

This report details the design of a comprehensive "Auto-Calibration" framework to bridge this semantic gap. The proposed system utilizes an active **Discovery Protocol** to probe the DAW, extracting semantic metadata (units, ranges, and curvature) via text analysis of the GUI feedback strings returned by the Live Object Model (LOM). It establishes a mathematical foundation for **Curve Fitting**—specifically focusing on the derivation of Skew Factors and Logarithmic coefficients from a "Three-Point Heuristic" sweep—and proposes a normalized **JSON Database** for persistent metadata storage. The result is a system capable of mathematically reconstructing the internal DSP transfer functions of black-box audio plugins, enabling precise, unit-aware control from an external Python agent.

2. Theoretical Framework of Parameter Mapping

To engineer a robust control framework, one must first deconstruct the mathematical and psychoacoustic principles that govern parameter design in audio software. The transfer

function $T(n)$ that converts the normalized input n to the physical value v is rarely identity or strictly linear. It is shaped by DSP constraints, interface ergonomics, and, most importantly, psychoacoustic laws.

2.1 The Psychoacoustics of Control

The design of audio parameters is heavily influenced by the Weber-Fechner law, which states that human sensation is logarithmic in relation to the physical stimulus intensity.⁷

- **Frequency Perception:** The human ear perceives pitch logarithmically. The interval between 100 Hz and 200 Hz (one octave) is perceived as equal in magnitude to the

interval between 1000 Hz and 2000 Hz (one octave).⁷ In a linear mapping of a frequency knob from 20 Hz to 20 kHz, the entire bass and lower-midrange spectrum (20 Hz – 2000 Hz)—which contains the fundamental frequencies of most musical instruments—would be compressed into the bottom 10% of the knob's travel. This would make fine-tuning a bass guitar equalizer impossible.

- **Amplitude Perception:** Similarly, loudness is perceived logarithmically and measured in decibels (dB). A linear fade from silence (0.0 amplitude) to unity gain (1.0 amplitude) results in a perception of the sound dropping off too abruptly near the silence point.⁸
- **Time Perception:** Parameters such as envelope decay or delay time often span orders of magnitude (e.g., 1 ms to 10 seconds). Linear control would make it difficult to dial in "snappy" values in the 10–500 ms range, which are common for percussion and rhythmic effects.

Consequently, DAWs and plugin frameworks apply "taper" or "skew" functions to these parameters to linearize the *perception* of change relative to the *rotation* of the control.

2.2 Mathematical Models of Parameter Curves

Three primary mathematical models describe the vast majority of audio parameter mappings found in VST, AU, and native Ableton devices: Linear, Exponential (Logarithmic), and Skewed Power Functions.

2.2.1 Linear Mapping

The linear map is the simplest case, used for parameters where physical magnitude scales uniformly with perception, or where the unit is a ratio or percentage. Common examples include "Dry/Wet" mix, "Pan" (in terms of control position), "Pulse Width," and "Resonance" (in some implementations).

Given a normalized input $n \in [0, 1]$, the physical value v is determined by the linear interpolation equation:

$$v = \min + n \cdot (\max - \min)$$

The inverse function, required for the agent to convert a target physical value back to a normalized float, is:

$$n = \frac{v - \min}{\max - \min}$$

This model requires only two data points to fully characterize: the minimum value (**min**) and the maximum value (**max**).⁴ For a Linear parameter, the normalized value **0.5** corresponds

exactly to the arithmetic mean of the range: $\frac{\min + \max}{2}$.

2.2.2 Logarithmic / Exponential Mapping

For frequency parameters, the standard mapping implies that a constant increment in the normalized value results in a constant *multiplication* of the physical value. This is mathematically defined as an exponential function.

$$v = \min \cdot \left(\frac{\max}{\min} \right)^n$$

This equation ensures that for every increment of n , the frequency multiplies by a constant factor. For example, if n increases by 0.1 , the frequency might double (increase by one octave), regardless of whether the current value is 100 Hz or 1000 Hz.

The inverse function, used to derive the normalized control value from a target frequency v , is logarithmic:

$$n = \frac{\ln(v) - \ln(\min)}{\ln(\max) - \ln(\min)}$$

This model assumes a strict logarithmic relationship. However, in practice, plugin developers often tweak this curve to provide even more resolution in specific areas, leading to the use of generalized skew factors.

2.2.3 The Skew Factor (Generalized Curvature)

A strict logarithmic map is sometimes too aggressive or not aggressive enough for specific parameters. The **JUCE** C++ framework, which powers a vast number of VST plugins and likely influences Ableton's internal device architecture, utilizes a "Skew Factor" to define continuous curves that can morph between linear, logarithmic, and exponential behaviors.¹¹

The NormalisableRange class in JUCE defines the relationship using a skew factor S . When $S = 1$, the mapping is linear. When $S \neq 1$, the mapping distorts. The underlying mathematics of the JUCE implementation, as revealed in developer discussions and documentation, often relies on applying an exponent to the proportion of the range.¹²

A generalized **Power Function** approximation that is robust for external control is:

$$v = \min + (\max - \min) \cdot n^\alpha$$

Where α is the curvature coefficient (or skew exponent).

- If $\alpha = 1$, the function is Linear.
- If $\alpha > 1$, the curve is exponential (concave up). This provides high resolution at the bottom of the range (e.g., 0%–50% of the knob covers 0–10% of the value range), which is ideal for parameters like Amp/Drive or Envelope Attack.
- If $\alpha < 1$, the curve is logarithmic (concave down). This provides high resolution at the top of the range.

The Critical "Midpoint" Theorem:

The presence of the exponent α introduces a third unknown. We cannot solve for the curve using only **min** and **max**. We require a third data point. By convention, the most informative point is the center of the control range, where $n = 0.5$.

If we know the physical value V_{mid} at $n = 0.5$, we can solve for α .

$$V_{mid} = \min + (\max - \min) \cdot 0.5^\alpha$$

Rearranging to solve for α :

$$\frac{V_{mid} - \min}{\max - \min} = 0.5^\alpha$$

$$\alpha = \log_{0.5} \left(\frac{V_{mid} - \min}{\max - \min} \right)$$

This **Three-Point Heuristic** ($V_{min}, V_{mid}, V_{max}$) forms the mathematical backbone of the Discovery Protocol. By empirically measuring these three values, the agent can analytically derive the exact curvature used by the plugin developer, regardless of the underlying code.⁷

3. The Discovery Protocol

The "Auto-Calibration" framework relies on a Discovery Protocol that runs during an initialization or "scan" phase. This protocol dialogues with Ableton Live via the **AbletonOSC** server, a MIDI Remote Script that exposes the Live Object Model (LOM) over the network.¹⁵

3.1 System Architecture

The architecture comprises three distinct entities interacting asynchronously:

1. **The Agent (Python):** The external brain. It orchestrates the sweep, parses the data, performs the curve fitting, and stores the results.
2. **The Bridge (AbletonOSC):** A server running inside Ableton's Python 3 environment. It listens on UDP port 11000 and broadcasts responses on port 11001. It acts as the interface to the LOM.¹⁷
3. **The Target (DeviceParameter):** The specific LOM object being profiled (e.g., "Frequency" on "Auto Filter").¹⁸

3.2 The "Three-Point Sweep" Algorithm

To profile a parameter without prior knowledge, the Agent performs a destructive read (or "sweep").

Warning: This process involves setting parameters to extreme values. To prevent audio damage, the Agent should mute the track or the master output before commencing the sweep.

Step 1: Handshake and ID Resolution

The agent first queries the device tree to locate the parameter index.

- **Query:** /live/device/get/parameters/name
- **Response:** (track_id, device_id, "Frequency", "Resonance", "Filter Type",...)
The Agent parses this list and identifies that "Frequency" is at parameter index 3.

Step 2: Probe Minimum (0.0)

- **Action:** Send OSC message /live/device/set/parameter/value 3 0.0.
- **Wait:** The agent must pause to allow the OSC packet to traverse the network, the DAW to process the command on the message thread, the GUI to update the display string, and the LOM to become consistent. A safe latency buffer is typically 10–50ms.¹⁵
- **Query:** Send OSC message /live/device/get/parameter/value_string 3.
- **Receive:** The bridge returns a string argument, e.g., "26.0 Hz".
- **Data Extraction:** The agent parses this string to extract $V_{min} = 26.0$ and Unit = "Hz".

Step 3: Probe Maximum (1.0)

- **Action:** Send OSC message /live/device/set/parameter/value 3 1.0.
- **Wait:** 10–50ms latency buffer.
- **Query:** Send OSC message /live/device/get/parameter/value_string 3.
- **Receive:** The bridge returns "19.9 kHz".
- **Data Extraction:** The agent parses this string. Crucially, it must normalize the unit. It

converts "19.9 kHz" to "19900.0 Hz" to match the unit of V_{min} . $V_{max} = 19900.0$.

Step 4: Probe Midpoint (0.5)

This step is the differentiator that allows for curvature detection.

- **Action:** Send OSC message `/live/device/set/parameter/value 3 0.5`.
- **Wait:** 10–50ms latency buffer.
- **Query:** Send OSC message `/live/device/get/parameter/value_string 3`.
- **Receive:** The bridge returns "710 Hz".
- **Data Extraction:** $V_{mid} = 710.0$.

Step 5: State Restoration

Ideally, the agent should read the initial value of the parameter before starting the sweep and restore it after Step 4 to minimize disruption to the user's set.

3.3 Handling Discrete and Enumerated Parameters

Not all parameters are continuous floats. Some are discrete selectors (e.g., "Filter Type": Lowpass, Highpass, Bandpass, Notch). The heuristic must detect these to avoid calculating invalid skew factors.

The Agent checks the `is_quantized` LOM property for the parameter.¹⁸

- If `is_quantized == False`: Proceed with the Three-Point Sweep (Continuous mode).
- If `is_quantized == True`: The parameter is Discrete.

For Discrete parameters, the sweep strategy changes. The standard Three-Point Sweep is insufficient because it misses the internal states. The Agent may employ a "Linear Crawl" or utilize the `value_items` property if available in the specific API version.

- **Value Items:** Some LOM versions expose a list of all possible display strings for a quantized parameter (e.g., ``).
- **Linear Crawl:** If `value_items` is unavailable, the Agent steps through normalized values (e.g., 0.0, 0.1, 0.2...) and monitors the `value_string` for changes. This constructs a map of {"Lowpass": [0.0, 0.24], "Highpass": [0.25, 0.49],...}.

3.4 Handling Asynchronous Responses

The discovery process involves significant network I/O. Implementing this synchronously (blocking the main thread while waiting for UDP packets) is inefficient and prone to timeouts. The Python agent should utilize the `asyncio` library to handle OSC responses.²⁰

- **Event Loop:** The main sweep logic runs as a coroutine.
- **Future/Promise Pattern:** When a query is sent, the agent creates a Future object. The

OSC server listener, running on the same event loop, completes this Future when the corresponding message arrives. This ensures exact matching of requests to responses and handles the inherent latency of the bridge gracefully.

4. Signal Processing and Semantic Parsing Strategy

The raw strings returned by the DAW (e.g., "22.5 kHz", "-inf dB", "1/16") are "semantic" data—they carry meaning intended for humans. To be useful for mathematical modeling, they must be rigorously parsed and normalized into machine-readable floats and base units.

4.1 Regex Engineering for Audio Units

Python's `re` module is the engine for this extraction.²¹ A robust regex pattern must capture the numeric value (which may include negatives, decimals, or "inf") and the optional unit suffix, while ignoring whitespace.

Composite Regex Pattern:

Code snippet

```
^s*          # Start of string, ignore leading whitespace
([+\\-]?(?:inf|[\\d]+(?:\\.\\d+)?)) # Group 1: Value. Captures integers, floats, signs, and 'inf'
s*          # Optional whitespace between value and unit
([a-zA-Z%]+)? # Group 2: Unit. Captures alpha characters or % symbol
s*$        # End of string, ignore trailing whitespace
```

This pattern handles:

- "220.0 Hz" → ("220.0", "Hz")
- "-12.5 dB" → ("-12.5", "dB")
- "50%" → ("50", "%")
- "-inf dB" → ("-inf", "dB")

4.2 Unit Normalization Logic

To allow for valid arithmetic (e.g., calculating the midpoint), the framework must standardize units to a **Base SI Unit**.

4.2.1 Frequency (Base: Hz)

- **Input:** Hz \rightarrow Multiplier: **1.0**.
- **Input:** kHz \rightarrow Multiplier: **1000.0**.
- **Example:** If V_{min} is "20 Hz" and V_{max} is "20 kHz", the raw calculation $20 - 20 = 0$ is wrong. Normalization converts V_{max} to **20,000**. Now the range is **19,980** Hz.

4.2.2 Time (Base: Seconds)

- **Input:** ms \rightarrow Multiplier: **0.001**.
- **Input:** s (or "sec") \rightarrow Multiplier: **1.0**.
- **Example:** Delay times often switch units dynamically. A delay might show "500 ms" at the midpoint and "2.0 s" at the max. Normalization ensures the range is **0.5** to **2.0** seconds.

4.2.3 Amplitude (Base: dB or Linear Gain)

- **Input:** dB \rightarrow No conversion is usually applied *during parsing*, as the parameter is natively logarithmic. The agent operates in the dB domain.
- **Special Case:** "-inf". The parser traps "inf" and assigns a sentinel value. For a fader, "-inf" effectively means "silence." However, mathematically, $\log(0)$ is undefined. The framework must define a **Minimum Effective Value (MEV)**. If the display shows "-inf", the agent might probe slightly higher (e.g., at **0.01**) to find the "floor" value (e.g., -144 dB or -70 dB) and use that as the practical minimum for curve fitting.

4.2.4 Ratios and Percentages

- **Input:** % \rightarrow Divisor: **100.0** (to map 0-100 to 0.0-1.0) OR keep as 0-100 float depending on user preference.
- **Input:** 1:2, 2:1 \rightarrow These are parsed as ratios. The regex needs a specialized branch for the colon \therefore . 2:1 becomes float 2.0.

4.3 Handling Tempo-Synced Values

A significant edge case in DAWs is tempo-synced parameters (e.g., LFO Rate, Delay Time). These do not display floats but fractions: "1/16", "1/4", "1 Bar", "1/4T" (Triplet), "1/16D" (Dotted).

These cannot be parsed as continuous floats. They are, effectively, **Enumerated** states.

- **Strategy:** If the regex fails to find a standard number, or if the unit contains /, the parameter is flagged as `Quantized_Tempo`.

- **Mapping:** The agent stores the string literal map: {0.1: "1/32", 0.2: "1/16",...}. The user inputs "1/16", and the agent looks up the corresponding normalized float.

5. Algorithmic Curve Fitting

Once the data triplet $(V_{min}, V_{mid}, V_{max})$ is captured and normalized, the agent proceeds to the Curve Fitting phase. This is purely mathematical and occurs offline (after the sweep).

The goal is to determine which of the three models (Linear, Log, Skewed) best fits the data, and to derive the coefficients necessary to reconstruct that curve.

5.1 Step 1: Linearity Test

The agent first hypothesizes that the parameter is Linear.

It calculates the **Theoretical Linear Midpoint** (L_{mid}):

$$L_{mid} = \frac{V_{min} + V_{max}}{2}$$

It then compares this to the measured V_{mid} .

$$\Delta = |V_{mid} - L_{mid}|$$

$$\text{Tolerance} = (V_{max} - V_{min}) \times 0.01 \quad (1\% \text{ error margin})$$

- If $\Delta < \text{Tolerance}$, the parameter is classified as **LINEAR**. No further calculation is needed. The mapping is standard linear interpolation.

5.2 Step 2: Skew Factor Derivation

If the parameter is not linear, it is assumed to follow a **Power Law (Skewed)** distribution. As discussed in Section 2.2.3, the Skew Factor α (alpha) determines the curvature.

The governing equation is:

$$\frac{V_{mid} - V_{min}}{V_{max} - V_{min}} = 0.5^\alpha$$

To solve for α :

Let $NormalizedMid = \frac{V_{mid} - V_{min}}{V_{max} - V_{min}}$.

(This represents where the midpoint value sits proportionally within the physical range).

Taking the logarithm of both sides:

$$\ln(NormalizedMid) = \ln(0.5^\alpha)$$

$$\ln(NormalizedMid) = \alpha \cdot \ln(0.5)$$

$$\alpha = \frac{\ln(NormalizedMid)}{\ln(0.5)}$$

This derived α is the key to the entire framework. It captures the exact non-linearity of the parameter.

5.3 Bi-Directional Mapping Functions

With α , V_{min} , and V_{max} stored, the agent can now perform precise bi-directional mapping.

Forward Mapping (Physical \rightarrow Normalized)

Used when the agent wants to set the parameter to a specific value (e.g., "Set Freq to 2500 Hz").

Input: *val*. Output: *norm*.

$$norm = \left(\frac{val - V_{min}}{V_{max} - V_{min}} \right)^{\frac{1}{\alpha}}$$

Inverse Mapping (Normalized \rightarrow Physical)

Used when the agent reads a normalized value and wants to know what it means (e.g., "What is the current frequency?").

Input: *norm*. Output: *val*.

$$val = V_{min} + (V_{max} - V_{min}) \cdot norm^\alpha$$

5.4 Example Calculation: Filter Frequency

- Measured Data:

- $V_{min} = 26.0$ Hz
- $V_{max} = 19900.0$ Hz
- $V_{mid} = 710.0$ Hz (at $n = 0.5$)
- **Linearity Check:**
 - $L_{mid} = (26 + 19900)/2 = 9963$
 - $|710 - 9963| = 9253$. Huge deviation. Not Linear.
- **Alpha Calculation:**
 - $NormalizedMid = \frac{710-26}{19900-26} = \frac{684}{19874} \approx 0.0344$
 - $\alpha = \frac{\ln(0.0344)}{\ln(0.5)} = \frac{-3.369}{-0.693} \approx 4.86$

The Skew Factor is **4.86**. This high exponent confirms a highly exponential/logarithmic curve, allocating a massive amount of control resolution to the lower end of the frequency spectrum, matching psychoacoustic expectations.

6. The Normalization Database (JSON Schema)

The discovery process is resource-intensive and should not run every time the agent sets a parameter. The calibrated data is persisted in a JSON "Normalization Database." This allows the agent to load the profile instantly at runtime.

6.1 Schema Design

The schema uses a hierarchical structure to organize data by Plugin, Version, and Parameter Name. This ensures collision avoidance between different plugins that might share parameter names (e.g., "Frequency").

JSON

```
{
  "plugin_name": "Ableton Auto Filter",
  "plugin_version": "11.0",
  "timestamp": "2023-10-27T14:30:00Z",
  "parameters": {
    "Frequency": {
```

```

    "id": 3,
    "type": "Continuous",
    "range": {
      "min": 26.0,
      "max": 19900.0,
      "mid": 710.0
    },
    "unit": "Hz",
    "curve_model": "POWER_SKEW",
    "skew_coefficient": 4.86,
    "input_domain": "Logarithmic"
  },
  "Resonance": {
    "id": 4,
    "type": "Continuous",
    "range": {
      "min": 0.0,
      "max": 1.25,
      "mid": 0.625
    },
    "unit": "%",
    "curve_model": "LINEAR",
    "skew_coefficient": 1.0,
    "input_domain": "Linear"
  },
  "Filter Type": {
    "id": 1,
    "type": "Enumerated",
    "options": {
      "0.0": "Lowpass",
      "0.25": "Highpass",
      "0.5": "Bandpass",
      "0.75": "Notch"
    }
  }
}

```

6.2 Schema Fields Explained

- **range:** Stores the raw physical values detected during the 3-point sweep. These are the ground truth constants for the mapping formulas.
- **curve_model:** The deduced mathematical model (LINEAR, POWER_SKEW, or DISCRETE).

- **skew_coefficient:** The derived α value.
- **unit:** The base SI unit after normalization.
- **input_domain:** A hint for the Agent's higher-level logic. Even if the internal curve is a Skewed Power function, the user interface might want to present a Logarithmic slider. This field helps bridge the UI-to-Math gap.

6.3 Versioning and Hashing

Plugins are often updated. "Auto Filter" in Live 10 might have different ranges than in Live 11. The database should ideally include a hash of the plugin's parameter list or a version string to ensure the loaded map matches the loaded plugin instance. If a mismatch is detected, the Agent should trigger a re-calibration (Discovery Phase).

7. Python Implementation Architecture

The practical implementation of this framework requires a structured Python application.

7.1 Class Structure

ParameterMap Class:

Encapsulates the mathematical logic for a single parameter.

- **Attributes:** min, max, mid, exponent, unit.
- **Methods:**
 - `value_to_norm(physical_val)` -> float
 - `norm_to_value(norm_float)` -> float

DeviceCalibrator Class:

Manages the Discovery Protocol.

- **Methods:**
 - `calibrate_device(track_id, device_id)`: Orchestrates the sweep.
 - `_sweep_parameter(param_index)`: Async coroutine for the 3-point probe.
 - `_parse_response(string)`: Regex engine.
 - `save_to_db(filepath)`: JSON serialization.

AbletonAgent Class:

The main controller interface.

- **Attributes:** calibration_db, osc_client.
- **Methods:**
 - `set_parameter(track, device, param_name, value, unit)`: The high-level command.
 - Example: `set_parameter(1, 0, "Frequency", 2500, "Hz")`.

- Logic: Looks up "Frequency" in DB \rightarrow Converts 2500 Hz to normalized float using $\alpha \rightarrow$ Sends OSC.

7.2 AsyncIO and Error Handling

Given the networked nature of OSC, error handling is critical.

- **Timeouts:** The `_sweep_parameter` coroutine must implement a timeout (e.g., `asyncio.wait_for(future, timeout=2.0)`). If the DAW does not respond (e.g., if a modal dialog is open in Live blocking the API), the agent should throw a `CalibrationError` rather than hanging indefinitely.
- **Packet Loss:** UDP is unreliable. The agent should implement a retry mechanism for the Discovery Phase queries.
- **Floating Point Precision:** Normalized floats are typically 32-bit. The agent should round the final output to 4–6 decimal places to strictly adhere to the $[0.0, 1.0]$ bounds and avoid sending 1.0000001 which might be rejected by the LOM.

8. Case Studies and Evaluation

To validate the framework, we examine specific challenging parameter types often encountered in production.

8.1 Case A: The "Drive" Knob (Distortion)

- **Characteristics:** Range 0 dB to 24 dB.
- **Behavior:** Often has a "soft" start and "aggressive" end.
- **Sweep Data:** Min=0, Max=24, Mid=6.
- **Linear Mid:** 12.
- **Analysis:** $V_{mid}(6) < L_{mid}(12)$. This indicates a Logarithmic-style curve (concave down behavior in some contexts, or requiring an exponent < 1 depending on axis orientation).
- **Alpha:** $\alpha = \log_{0.5}((6 - 0)/(24 - 0)) = \log_{0.5}(0.25) = 2.0$.
- **Result:** A standard quadratic power curve fits this perfectly. The agent can now accurately set Drive to 18 dB, knowing it corresponds to a specific point on the quadratic curve.

8.2 Case B: The Synthesizer Envelope (ADSR)

- **Characteristics:** Attack Time. Range 1 ms to 20 s (20,000 ms).
- **Sweep Data:** Min=1, Max=20000, Mid=600.

- **Analysis:**
 - $L_{mid} \approx 10000$
 - $V_{mid} = 600$. Massive deviation.
 - $NormalizedMid = 599/19999 \approx 0.03$
 - $\alpha \approx 5.0$
- **Result:** The framework calculates a steep exponent of 5.0. This correctly models the behavior where the first half of the knob covers only the first 600ms (snappy attacks), and the second half covers the remaining 19.4 seconds (slow swells).

9. Conclusion

The "Auto-Calibration" framework effectively solves the Semantic Gap in OSC-based DAW control. By treating the DAW as a black box and using its own GUI feedback loop as the source of truth, the system effectively reverse-engineers the internal DSP scaling curves.

The **Three-Point Heuristic** coupled with the **Skewed Power Function** provides a mathematically rigorous method to approximate standard audio tapers used in JUCE and LOM frameworks. This approach decouples the agent from the specific implementation details of the VST, allowing for a universal "Driver" for digital audio generation.

This system empowers Python-based agents to move beyond random parameter perturbation and into the realm of intentional, semantic sound design—setting filters to musical intervals, compressors to precise thresholds, and delays to exact rhythmic subdivisions.

References

- ³: Normalization concepts in Python/Matplotlib.
- ¹⁵: AbletonOSC API documentation and architecture.
- ¹¹: JUCE NormalisableRange and Skew Factor math.
- ⁷: Psychoacoustics, Logarithmic scales in audio.
- ²¹: Regex for parsing audio units.
- ¹⁸: Live Object Model (LOM) DeviceParameter properties.

Works cited

1. Signal Processing (scipy.signal) — SciPy v1.17.0 Manual, accessed February 10, 2026, <https://docs.scipy.org/doc/scipy/tutorial/signal.html>
2. Live Instrument Reference — Ableton Reference Manual Version 12, accessed February 10, 2026, <https://www.ableton.com/en/manual/live-instrument-reference/>

3. Colormap normalization — Matplotlib 3.10.8 documentation, accessed February 10, 2026, <https://matplotlib.org/stable/users/explain/colors/colormapnorms.html>
4. How to normalize data to 0-1 range? - Cross Validated, accessed February 10, 2026, <https://stats.stackexchange.com/questions/70801/how-to-normalize-data-to-0-1-range>
5. Virtual Studio Technology Plug-In Specification 2.0 Software Development Kit - jVSTWrapper, accessed February 10, 2026, <https://jvstwrapper.sourceforge.net/vst20spec.pdf>
6. Scaling Curves - Advanced Usage - Gig Performer®, accessed February 10, 2026, <https://gigperformer.com/scaling-curves-advanced-usage>
7. The non-linearities of the Human Ear - AudioCheck.net, accessed February 10, 2026, https://www.audiocheck.net/soundtests_nonlinear.php
8. Linear and Logarithmic Scaling - Rational Acoustics, accessed February 10, 2026, <https://support.rationalacoustics.com/support/solutions/articles/150000214511-linear-and-logarithmic-scaling>
9. Frequency Warping in the Design and - Implementation of Fixed-Point Audio - DSP Concepts, accessed February 10, 2026, <https://dspconcepts.com/sites/default/files/warp-conference.pdf>
10. Convert a number range to another range, maintaining ratio - Stack Overflow, accessed February 10, 2026, <https://stackoverflow.com/questions/929103/convert-a-number-range-to-another-range-maintaining-ratio>
11. juce::NormalisableRange< ValueType > Class Template Reference, accessed February 10, 2026, https://docs.juce.com/master/classjuce_1_1NormalisableRange.html
12. Frequency parameter / slider log scaling - JUCE Forum, accessed February 10, 2026, <https://forum.juce.com/t/frequency-parameter-slider-log-scaling/18258>
13. How to un-skew slider values? - JUCE Forum, accessed February 10, 2026, <https://forum.juce.com/t/how-to-un-skew-slider-values/52962>
14. Logarithmic Sliders - JUCE Step by step - WordPress.com, accessed February 10, 2026, <https://jucestepbystep.wordpress.com/logarithmic-sliders/>
15. AbletonOSC: A unified control API for Ableton Live - New Interfaces for Musical Expression, accessed February 10, 2026, https://nime.org/proceedings/2023/nime2023_60.pdf
16. ideofoms/AbletonOSC: Control Ableton Live via Open Sound Control (OSC) - GitHub, accessed February 10, 2026, <https://github.com/ideofoms/AbletonOSC>
17. AbletonOSC - Sam Twidale - GitLab, accessed February 10, 2026, <https://gitlab.com/OnikaStudio/AbletonOSC>
18. DeviceParameter - Live Object Model - Max Documentation - Cycling '74, accessed February 10, 2026, <https://docs.cycling74.com/apiref/lom/deviceparameter/>
19. LOM devices parameters value unit - Misc Forum - Cycling '74, accessed February 10, 2026, <https://cycling74.com/forums/lom-devices-parameters-value-unit>
20. Server — python-osc 1.7.1 documentation, accessed February 10, 2026,

<https://python-osc.readthedocs.io/en/latest/server.html>

21. re — Regular expression operations — Python 3.14.3 documentation, accessed February 10, 2026, <https://docs.python.org/3/library/re.html>
22. Using regex in python to parse log files to extract values - Stack Overflow, accessed February 10, 2026, <https://stackoverflow.com/questions/76059766/using-regex-in-python-to-parse-log-files-to-extract-values>
23. Parsing units with javascript regex - Stack Overflow, accessed February 10, 2026, <https://stackoverflow.com/questions/31330090/parsing-units-with-javascript-regex>