MMMAudio Documentation

DSP library for Python and Mojo

Sam Pluta

Table of Contents

1. MMMAudio Documentation

Welcome to the MMMAudio documentation! MMMAudio is a high-performance audio processing library that combines the ease of Python with the speed of Mojo for real-time audio applications.

1.1 Features

- **High Performance**: Leverages Mojo's SIMD capabilities for optimal audio processing
- **Dual Language Support**: Write audio logic in Python, optimize critical paths in Mojo
- **Real-time Capable**: Designed for low-latency audio applications
- **Modular Design**: Composable DSP building blocks
- **ML Integration**: Support for neural network audio processing

1.2 Quick Start

```
from mmm_src.MMMAudio import MMMAudio
from mmm_dsp.Osc import SineOsc

# Create audio engine
audio = MMMAudio(sample_rate=44100, buffer_size=512)

# Create and connect oscillator
osc = SineOsc(frequency=440.0)
audio.connect(osc, audio.output)

# Start processing
audio.start()
```

1.3 Documentation Structure

- **Getting Started**: Installation and basic usage
- **API Reference**: Complete API documentation
- **Examples**: Practical usage examples
- **Development**: Contributing and development guide

1.4 Architecture

MMMAudio is built around a graph-based processing model where audio flows through connected nodes. Each node can be implemented in either Python (for flexibility) or Mojo (for performance).

1.4.1 Core Components

- **DSP Modules** (`mmm_dsp`): Basic audio processing building blocks
- **Framework** (`mmm_src`): Audio engine and graph management
- **Utilities** (`mmm_utils`): Mathematical and utility functions

1.5 Community

- **GitHub**: https://github.com/tedmoore/MMMAudio
- **Issues**: Report bugs and feature requests on GitHub

2. Getting Started with MMMAudio

MMMAudio is a high-performance audio processing library that combines Python's ease of use with Mojo's performance for real-time audio applications.

2.1 Installation

2.1.1 Prerequisites

- Python 3.9 or higher
- Mojo compiler (latest version)
- Audio drivers (ASIO on Windows, CoreAudio on macOS, ALSA on Linux)

2.1.2 Install from Source

1. Clone the repository:

```
git clone https://github.com/tedmoore/MMMAudio.git
cd MMMAudio
```

1. Install Python dependencies:

```
pip install -r requirements.txt
```

1. Verify Mojo installation:

```
mojo --version
```

2.2 Quick Start

2.2.1 Basic Audio Setup

Create a simple audio processing chain:

```
from mmm_src.MMMAudio import MMMAudio
from mmm_dsp.Osc import SineOsc

# Initialize audio engine
audio = MMMAudio(
    sample_rate=44100,
    buffer_size=512,
    channels=2
)

# Create a sine wave oscillator
osc = SineOsc(frequency=440.0, amplitude=0.5)

# Connect oscillator to audio output
audio.connect(osc, audio.output)

# Start audio processing
audio.start()

# Let it run for 5 seconds
import time
time.sleep(5)

# Stop audio
audio.stop()
```

2.2.2 Using Mojo for Performance

For performance-critical operations, use Mojo implementations:

```
from mmm_utils.functions import linlin
from algorithm import parallelize

# Process control data with SIMD optimization
midi_velocities = [64, 80, 96, 127]  # MIDI velocity values
gains = []

for velocity in midi_velocities:
    # Convert MIDI velocity to linear gain using Mojo function
    gain = linlin(float(velocity), 0.0, 127.0, 0.0, 1.0)
    gains.append(gain)

print(f"Converted gains: {gains}")
```

2.3 Core Concepts

2.3.1 Audio Graph

MMMAudio uses a graph-based processing model where audio flows through connected nodes:

```
# Create nodes
input_node = audio.input
osc1 = SineOsc(440.0)
osc2 = SineOsc(880.0)
mixer = Mixer(2)  # 2-input mixer
output_node = audio.output

# Connect the graph
audio.connect(osc1, mixer.input[0])
audio.connect(osc2, mixer.input[1])
audio.connect(mixer, output_node)
```

2.3.2 SIMD Optimization

Mojo functions support SIMD operations for processing multiple values simultaneously:

```
# Process 4 frequencies at once
from mmm_utils.functions import midicps

midi_notes = SIMD[DType.float64, 4](60.0, 64.0, 67.0, 72.0)  # C major chord
frequencies = midicps[4](midi_notes)  # Convert to frequencies
```

2.3.3 Real-time Processing

MMMAudio is designed for real-time audio with low latency:

```
# Configure for low latency
audio = MMMAudio(
    sample_rate=44100,
    buffer_size=128,  # Small buffer for low latency
    channels=2
)

# Use efficient processing chains
reverb = Reverb(room_size=0.5, damping=0.7)
audio.connect(audio.input, reverb)
audio.connect(reverb, audio.output)
```

2.4 Examples

2.4.1 Simple Synthesizer

```
from mmm_src.MMMAudio import MMMAudio
from mmm_dsp.Osc import SineOsc
from mmm_dsp.Env import ADSR
from mmm_dsp.Filters import LowPass

# Create audio engine
audio = MMMAudio()

# Create synthesis components
osc = SineOsc(frequency=440.0)
envelope = ADSR(attack=0.1, decay=0.2, sustain=0.7, release=0.5)
filter = LowPass(cutoff=2000.0, resonance=0.5)
```

```
# Build signal chain
audio.connect(osc, filter)
audio.connect(envelope, filter.cutoff_mod)  # Envelope modulates filter
audio.connect(filter, audio.output)

# Start synthesis
audio.start()
envelope.trigger()  # Trigger note

time.sleep(2)
envelope.release()  # Release note
time.sleep(1)

audio.stop()
```

2.4.2 Multi-channel Processing

```
# Stereo processing with different effects per channel
audio = MMMAudio(channels=2)

# Create stereo sources
osc_left = SineOsc(440.0)
osc_right = SineOsc(442.0)  # Slightly detuned for stereo effect

# Different processing per channel
delay_left = Delay(time=0.3, feedback=0.3)
delay_right = Delay(time=0.4, feedback=0.2)

# Connect stereo processing
audio.connect(osc_left, delay_left)
audio.connect(osc_right, delay_right)
audio.connect(delay_left, audio.output.left)
audio.connect(delay_right, audio.output.right)

audio.start()
```

2.5 Performance Tips

2.5.1 Use SIMD When Possible

```
# Instead of processing one value at a time:
for i in range(len(values)):
    result[i] = linlin(values[i], 0.0, 1.0, 20.0, 20000.0)

# Process multiple values with SIMD:
from mmm_utils.functions import linlin

# Convert list to SIMD and process all at once
values_simd = SIMD[DType.float64, 4].from_list(values[:4])
results_simd = linlin[4](values_simd, 0.0, 1.0, 20.0, 20000.0)
```

2.5.2 Optimize Buffer Sizes

```
# Balance latency vs CPU usage
audio = MMMAudio(
    buffer_size=256,  # Good balance for most applications
    sample_rate=44100
)

# For very low latency (may increase CPU usage):
audio_low_latency = MMMAudio(buffer_size=64)

# For maximum efficiency (higher latency):
audio_efficient = MMMAudio(buffer_size=1024)
```

2.5.3 Reuse Objects

```
# Create objects once and reuse
osc = SineOsc()

# Change parameters instead of creating new objects
osc.set_frequency(880.0)
osc.set_amplitude(0.5)

# This is more efficient than:
# osc = SineOsc(frequency=880.0, amplitude=0.5)  # Creates new object
```

2.6 Troubleshooting

2.6.1 Audio Dropouts

If you experience audio dropouts:

1. Increase buffer size:

```
audio = MMMAudio(buffer_size=512)  # or higher
```

2. Reduce processing complexity in real-time callbacks
3. Use Mojo implementations for CPU-intensive operations

2.6.2 Import Errors

Make sure the project is in your Python path:

```
import sys
sys.path.append('/path/to/MMMAudio')
```

Or install in development mode:

```
pip install -e .
```

2.6.3 Mojo Compilation Issues

Ensure you have the latest Mojo compiler:

```
mojo --version
```

Update if necessary according to Mojo documentation.

2.7 Next Steps

- Explore the API Reference for detailed function documentation
- Check out Examples for more complex usage patterns
- Read the Development Guide to contribute

2.8 Community and Support

- **GitHub Issues**: Bug reports and feature requests
- **Discussions**: Questions and community interaction
- **Documentation**: Complete API reference and guides

Happy audio processing with MMMAudio!

3. API Reference

3.1 API Reference

This section contains the complete API reference for MMMAudio, organized by module.

3.1.1 Core DSP (`mmm_dsp`)

The core DSP modules provide the fundamental building blocks for audio processing:

- **Utilities**: Mathematical utility functions
- **Oscillators**: Sine, square, sawtooth, and noise generators
- **Filters**: Low-pass, high-pass, band-pass filters
- **Envelopes**: ADSR and other envelope generators
- **Delays**: Delay lines and echo effects
- **Buffers**: Audio buffer management
- **Effects**: Distortion and saturation

3.1.2 Framework (`mmm_src`)

The framework modules handle audio engine management and processing:

- **Audio Engine**: Main audio processing engine
- **Graph System**: Audio graph management
- **Traits**: Core interfaces and traits
- **Scheduler**: Event scheduling and timing

3.1.3 Utilities (`mmm_utils`)

Utility modules provide mathematical functions and helpers:

- **Functions**: Mathematical utility functions
- **FFT**: Fast Fourier Transform utilities
- **Windows**: Window functions for DSP

3.1.4 Documentation Conventions

Function Signatures

Functions are documented with their complete signatures including type information:

```
fn linlin[N: Int = 1](
    value: SIMD[DType.float64, N],
    in_min: SIMD[DType.float64, N],
    in_max: SIMD[DType.float64, N],
    out_min: SIMD[DType.float64, N],
    out_max: SIMD[DType.float64, N]
) -> SIMD[DType.float64, N]
```

SIMD Support

Most functions support SIMD operations for processing multiple values simultaneously. The `N` parameter controls the SIMD width.

Examples

Each function includes practical examples showing typical usage patterns.

3.2 Core DSP

3.2.1 functions

MMM Utility Functions

This module provides essential utility functions for audio processing and mathematical operations in the MMMAudio framework. All functions are optimized for SIMD operations to achieve maximum performance on modern processors.

The functions in this module include: - Range mapping functions (linear and exponential) - Clipping and wrapping utilities - Interpolation algorithms - MIDI/ frequency conversion - Audio utility functions - Random number generation

All functions support vectorized operations through SIMD types for processing multiple values simultaneously.

Functions

LINLIN

Maps values from one range to another range linearly.

This function performs linear mapping from an input range to an output range. Values outside the input range are clamped to the corresponding output boundaries. This is commonly used for scaling control values, normalizing data, and converting between different parameter ranges.

Examples:

```
# Map MIDI velocity (0-127) to gain (0.0-1.0)
velocity = SIMD[DType.float64, 1](64.0)
gain = linlin(velocity, 0.0, 127.0, 0.0, 1.0)  # Returns 0.504

# Map multiple control values simultaneously
controls = SIMD[DType.float64, 4](0.25, 0.5, 0.75, 1.0)
frequencies = linlin[4](controls, 0.0, 1.0, 20.0, 20000.0)

# Invert a normalized range
normal_vals = SIMD[DType.float64, 2](0.3, 0.7)
inverted = linlin[2](normal_vals, 0.0, 1.0, 1.0, 0.0)
```

Signature

```
linlin[N: Int = 1](value: SIMD[DType.float64, N], in_min: SIMD[DType.float64, N], in_max: SIMD[DType.float64, N], out_min: SIMD[DType.float64, N], out_max: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int` - Size of the SIMD vector (defaults to 1).

Arguments

- **value**: `SIMD` - The values to map.- **in_min**: `SIMD` - The minimum of the input range.- **in_max**: `SIMD` - The maximum of the input range.- **out_min**: `SIMD` - The minimum of the output range.- **out_max**: `SIMD` - The maximum of the output range.

Returns

Type: `SIMD` The linearly mapped values in the output range.

LINEXP

Maps values from one linear range to another exponential range.

This function performs exponential mapping from a linear input range to an exponential output range. This is essential for musical applications where frequency perception is logarithmic. Both output range values must be positive.

Examples:

```
# Map linear slider (0-1) to frequency range (20Hz-20kHz)
slider_pos = SIMD[DType.float64, 1](0.5)
```

```
frequency = linexp(slider_pos, 0.0, 1.0, 20.0, 20000.0)  # ≈ 632 Hz

# Map MIDI controller to filter cutoff frequencies
cc_values = SIMD[DType.float64, 4](0.0, 0.33, 0.66, 1.0)
cutoffs = linexp[4](cc_values, 0.0, 1.0, 100.0, 10000.0)

# Create exponential envelope shape
linear_time = SIMD[DType.float64, 1](0.8)
exp_amplitude = linexp(linear_time, 0.0, 1.0, 0.001, 1.0)
```

Signature

```
linexp[N: Int = 1](value: SIMD[DType.float64, N], in_min: SIMD[DType.float64, N], in_max: SIMD[DType.float64, N], out_min: SIMD[DType.float64, N], out_max: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int` - Size of the SIMD vector (defaults to 1).

Arguments

- **value**: `SIMD` - The values to map.- **in_min**: `SIMD` - The minimum of the input range.- **in_max**: `SIMD` - The maximum of the input range.- **out_min**: `SIMD` - The minimum of the output range (must be > 0).- **out_max**: `SIMD` - The maximum of the output range (must be > 0).

Returns

Type: `SIMD` The exponentially mapped values in the output range.

CLIP

Clips each element in the SIMD vector to the specified range. Parameters: N: size of the SIMD vector - defaults to 1 Args: val: The SIMD vector to clip. Each element will be clipped individually. lo: The minimum value. hi: The maximum value. Returns: The clipped SIMD vector.

Signature

```
clip[N: Int = 1](val: SIMD[DType.float64, N], lo: SIMD[DType.float64, N], hi: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int`

Arguments

- **val**: `SIMD` - **lo**: `SIMD` - **hi**: `SIMD`

Returns

Type: `SIMD`

WRAP

Wraps a value around a specified range. Parameters: N: size of the SIMD vector - defaults to 1.

Signature

```
wrap[N: Int = 1](value: SIMD[DType.float64, N], min_val: SIMD[DType.float64, N], max_val: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int`

Arguments

- **value**: `SIMD` - The value to wrap.- **min_val**: `SIMD` - The minimum of the range.- **max_val**: `SIMD` - The maximum of the range.

Returns

Type: `SIMD`

Wraps a value around a specified range.

Signature

```
wrap[N: Int = 1](value: SIMD[DType.int64, N], min_val: SIMD[DType.int64, N], max_val: SIMD[DType.int64, N]) -> SIMD[DType.int64, N]
```

Parameters

- **N**: `Int` - Size of the SIMD vector - defaults to 1.

Arguments

- **value**: `SIMD` - The value to wrap.- **min_val**: `SIMD` - The minimum of the range.- **max_val**: `SIMD` - The maximum of the range.

Returns

Type: `SIMD`

QUADRATIC_INTERP

Performs quadratic interpolation between three points.

Signature

```
quadratic_interp(y0: Float64, y1: Float64, y2: Float64, x: Float64) -> Float64
```

Arguments

- **y0**: `Float64` - The value at position 0.- **y1**: `Float64` - The value at position 1.- **y2**: `Float64` - The value at position 2.- **x**: `Float64` - The interpolation position (typically between 0 and 2).

Returns

Type: `Float64` The interpolated value at position x.

CUBIC_INTERP

Performs cubic interpolation between.

Cubic Intepolation equation from *The Audio Programming Book* by Richard Boulanger and Victor Lazzarini. pg. 400

Signature

```
cubic_interp(p0: Float64, p1: Float64, p2: Float64, p3: Float64, t: Float64) -> Float64
```

Arguments

- **p0**: `Float64` - Point to the left of p1.- **p1**: `Float64` - Point to the left of the float t.- **p2**: `Float64` - Point to the right of the float t.- **p3**: `Float64` - Point to the right of p2.- **t**: `Float64` - Interpolation parameter (0.0 to 1.0).

Returns

Type: `Float64` Interpolated value.

LAGRANGE4

Perform Lagrange interpolation for 4th order case (from JOS Faust Model). This is extrapolated from the JOS Faust filter model.

lagrange4N -> SIMD[Float64, N]

Signature

```
lagrange4[N: Int = 1](sample0: SIMD[DType.float64, N], sample1: SIMD[DType.float64, N], sample2: SIMD[DType.float64, N], sample3: SIMD[DType.float64, N], samp
le4: SIMD[DType.float64, N], frac: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int` - Size of the SIMD vector - defaults to 1.

Arguments

- **sample0**: `SIMD` - The first sample.- **sample1**: `SIMD` - The second sample.- **sample2**: `SIMD` - The third sample.- **sample3**: `SIMD` - The fourth sample.- **sample4**: `SIMD` - The fifth sample.- **frac**: `SIMD` - The fractional delay (0.0 to 1.0) which is the location between sample0 and sample1.

Returns

Type: `SIMD` The interpolated value.

LERP

Performs linear interpolation between two points.

lerpN -> Float64 or SIMD[Float64, N]

Signature

```
lerp[N: Int = 1](p0: SIMD[DType.float64, N], p1: SIMD[DType.float64, N], t: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int` - Size of the SIMD vector - defaults to 1.

Arguments

- **p0**: `SIMD` - The starting point.- **p1**: `SIMD` - The ending point.- **t**: `SIMD` - The interpolation parameter (0.0 to 1.0).

Returns

Type: `SIMD` The interpolated value.

MIDICPS

Signature

```
midicps(midi_note_number: Int64, reference_midi_note: Int64 = 69, reference_frequency: Float64 = 440) -> Float64
```

Arguments

- **midi_note_number**: `Int64` - **reference_midi_note**: `Int64 = 69` - **reference_frequency**: `Float64 = 440`

Returns

Type: `Float64`

Signature

```
midicps(midi_note_number: Float64, reference_midi_note: Float64 = 69, reference_frequency: Float64 = 440) -> Float64
```

Arguments

- **midi_note_number**: `Float64` - **reference_midi_note**: `Float64 = 69` - **reference_frequency**: `Float64 = 440`

Returns

Type: `Float64`

CPSMIDI

Signature

```
cpsmidi(freq: Float64, reference_midi_note: Float64 = 69, reference_frequency: Float64 = 440) -> Float64
```

Arguments

- **freq**: `Float64` - **reference_midi_note**: `Float64 = 69` - **reference_frequency**: `Float64 = 440`

Returns

Type: `Float64`

MIX

Signature

```
mix(mut output: List[Float64], *lists: List[Float64])
```

Arguments

- **output**: `List` - ***lists**: `List`

SANITIZE

Signature

```
sanitize[N: Int = 1](x: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int`

Arguments

- **x**: `SIMD`

Returns

Type: `SIMD`

RANDOM_EXP_FLOAT64

Generates a random float64 value from an exponential distribution.

Signature

```
random_exp_float64[N: Int = 1](min: SIMD[DType.float64, N], max: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int` - Size of the SIMD vector - defaults to 1.

Arguments

- **min**: `SIMD` - The minimum value (inclusive).- **max**: `SIMD` - The maximum value (inclusive).

Returns

Type: `SIMD`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.2 Buffer

BUFFER

Has 2 possible constructors:

1) Buffer(lists: List[List[Float64]], buf_sample_rate: Float64 = 48000.0). - lists: List of channels, each channel is a List of Float64 samples. - buf_sample_rate: Sample rate of the buffer (default is 48000.0).

2) Buffer(num_chans: Int64 = 2, samples: Int64 = 48000, buf_sample_rate: Float64 = 48000.0). - num_chans: Number of channels (default is 2 for stereo). - samples: Number of samples per channel (default is 48000 for 1 second at 48kHz). - buf_sample_rate: Sample rate of the buffer (default is 48000.0).

Parent Traits: `AnyType`, `Buffable`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.3 Functions

`fn` get_num_frames

Return the number of frames in the buffer.

SIGNATURE

```
get_num_frames(self) -> Float64
```

RETURNS

Type: `Float64`

`fn` get_duration

Return the duration of the buffer in seconds.

SIGNATURE

```
get_duration(self) -> Float64
```

RETURNS

Type: `Float64`

`fn` get_buf_sample_rate

Return the sample rate of the buffer.

SIGNATURE

```
get_buf_sample_rate(self) -> Float64
```

RETURNS

Type: `Float64`

`fn` quadratic_interp_loc

SIGNATURE

```
quadratic_interp_loc(self, idx: Int64, idx1: Int64, idx2: Int64, frac: Float64, chan: Int64) -> Float64
```

ARGUMENTS

- **idx**: `Int64`
- **idx1**: `Int64`
- **idx2**: `Int64`
- **frac**: `Float64`
- **chan**: `Int64`

RETURNS

Type: `Float64`

`fn` linear_interp_loc

SIGNATURE

```
linear_interp_loc(self, idx: Int64, idx1: Int64, frac: Float64, chan: Int64) -> Float64
```

ARGUMENTS

- **idx**: `Int64`
- **idx1**: `Int64`
- **frac**: `Float64`
- **chan**: `Int64`

RETURNS

Type: `Float64`

`fn` read_sinc

SIGNATURE

```
read_sinc(mut self, chan: Int64, phase: Float64, last_phase: Float64) -> Float64
```

ARGUMENTS

- **chan**: `Int64`
- **phase**: `Float64`
- **last_phase**: `Float64`

RETURNS

Type: `Float64`

`fn` read

A read operation on the buffer that reads a multichannel buffer and returns a SIMD vector of size N. It will start reading from the channel specified by start_chan and read N channels from there.read(start_chan, phase, interp=0)

SIGNATURE

```
read[N: Int = 1](mut self, start_chan: Int64, phase: Float64, interp: Int64 = 0) -> SIMD[DType.float64, N]
```

PARAMETERS

- **N**: `Int` - The number of channels to read (default is 1). The SIMD vector returned will have this size as well.

ARGUMENTS

- **start_chan**: `Int64` - The starting channel index to read from (0-based).
- **phase**: `Float64` - The phase position to read from, where 0.0 is the start of the buffer and 1.0 is the end.
- **interp**: `Int64 = 0` - The interpolation method to use (0 = linear, 1 = quadratic).

RETURNS

Type: `SIMD`

`fn` write

SIGNATURE

```
write[N: Int = 1](mut self, value: SIMD[DType.float64, N], index: Int64, start_channel: Int64 = 0)
```

PARAMETERS

- **N**: `Int`

ARGUMENTS

- **value**: `SIMD`
- **index**: `Int64`
- **start_channel**: `Int64 = 0`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.4 Osc

PHASOR

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.5 Functions

`fn` increment_phase

SIGNATURE

```
increment_phase(mut self: Phasor[N], freq: SIMD[DType.float64, N], os_index: Int = 0)
```

ARGUMENTS

- **freq**: `SIMD`
- **os_index**: `Int` = `0`

`fn` next

SIGNATURE

```
next(mut self: Phasor[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, os_index: Int = 0) -> SIMD[DType.fl
oat64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### Osc

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.6 Functions

`fn` next

SIGNATURE

```
next(mut self: Osc[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, osc_type: SIMD[DType.int64, N] = 0, in
terp: SIMD[DType.int64, N] = 1, os_index: Int = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`

- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **osc_type**: `SIMD` = `0`
- **interp**: `SIMD` = `1`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

`fn` next_interp

SIGNATURE

```
next_interp(mut self: Osc[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, osc_types: List[Int64] = List[Int64, False](0, 4, 5, 6, Tuple[]()), osc_frac: SIMD[DType.float64, N] = 0, interp: Int64 = 1, os_index: Int = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **osc_types**: `List` = `List[Int64, False](0, 4, 5, 6, Tuple[]())`
- **osc_frac**: `SIMD` = `0`
- **interp**: `Int64` = `1`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### SinOsc

A sine wave oscillator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.7 Functions

`fn` next

SIGNATURE

```
next(mut self: SinOsc[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, interp: Int64 = 0, os_index: Int = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **interp**: `Int64` = `0`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### LFSaw

A low-frequency sawtooth oscillator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.8 Functions

`fn` next

SIGNATURE

```
next(mut self: LFSaw[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, interp: Int64 = 0, os_index: Int =
0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **interp**: `Int64` = `0`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### LFSquare

A low-frequency square wave oscillator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.9 Functions

`fn` next

SIGNATURE

```
next(mut self: LFSquare[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, interp: Int64 = 0, os_index: Int
= 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **interp**: `Int64` = `0`

- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### LFTri

A low-frequency triangle wave oscillator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.10 Functions

`fn` next

SIGNATURE

```
next(mut self: LFTri[N], freq: SIMD[DType.float64, N] = 100, phase_offset: SIMD[DType.float64, N] = 0, trig: Float64 = 0, interp: Int64 = 0, os_index: Int =
0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **phase_offset**: `SIMD` = `0`
- **trig**: `Float64` = `0`
- **interp**: `Int64` = `0`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### Impulse

An oscillator that generates an impulse signal. Arguments: world_ptr: Pointer to the MMMWorld instance.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.11 Functions

`fn` next

Generate the next impulse sample.

SIGNATURE

```
next(mut self: Impulse[N], freq: SIMD[DType.float64, N] = 100, trig: Float64 = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **trig**: `Float64` = `0`

RETURNS

Type: `SIMD`

`fn` **get_phase**

SIGNATURE

```
get_phase(mut self: Impulse[N]) -> SIMD[DType.float64, N]
```

RETURNS

Type: `SIMD`

---### Dust

A low-frequency dust noise oscillator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.12 Functions

`fn` **next**

Generate the next dust noise sample.

SIGNATURE

```
next(mut self: Dust[N], freq: SIMD[DType.float64, N] = 100, trig: Float64 = 1) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **freq**: `SIMD` = `100`
- **trig**: `Float64` = `1`

RETURNS

Type: `SIMD`

`fn` **next_range**

Generate the next dust noise sample.

SIGNATURE

```
next_range(mut self: Dust[N], low: SIMD[DType.float64, N] = 100, high: SIMD[DType.float64, N] = 2000, trig: Float64 = 1) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **low**: `SIMD` = `100`
- **high**: `SIMD` = `2000`
- **trig**: `Float64` = `1`

RETURNS

Type: `SIMD`

`fn` **get_phase**

SIGNATURE

```
get_phase(mut self: Dust[N]) -> SIMD[DType.float64, N]
```

RETURNS

Type: `SIMD`

---### LFNoise

Low-frequency noise oscillator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.13 Functions

`fn` **next**

Generate the next low-frequency noise sample.

SIGNATURE

```
next(mut self, freq: Float64 = 100, interp: Int64 = 0) -> Float64
```

ARGUMENTS

- **freq**: `Float64` = `100`
- **interp**: `Int64` = `0`

RETURNS

Type: `Float64`

---### Sweep

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.14 Functions

`fn` **increment_phase**

SIGNATURE

```
increment_phase(mut self, freq: Float64)
```

ARGUMENTS

- **freq**: `Float64`

`fn` **next**

SIGNATURE

```
next(mut self, freq: Float64 = 100, trig: Float64 = 0) -> Float64
```

ARGUMENTS

- **freq**: `Float64` = `100`
- **trig**: `Float64` = `0`

RETURNS

Type: `Float64`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.15 Filters

Functions

TF2S

Signature

```
tf2s[N: Int = 1](coeffs: List[SIMD[DType.float64, N]], mut coeffs_out: List[SIMD[DType.float64, N]], sample_rate: Float64)
```

Parameters

- **N**: `Int`

Arguments

- **coeffs**: `List` - **coeffs_out**: `List` - **sample_rate**: `Float64`

LAG

A lag processor that smooths input values over time based on a specified lag time in seconds.

Arguments:

```
**N:** Number of channels Lag will process. (This creates SIMD parallel processing.)
```

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.16 Functions

`fn` get_small_simd

SIGNATURE

```
get_small_simd(mut self, in_samp: SIMD[DType.float64, N], j: Int)
```

ARGUMENTS

- **in_samp**: `SIMD`
- **j**: `Int`

`fn` put_small_simd

SIGNATURE

```
put_small_simd(mut self, j: Int)
```

ARGUMENTS

- **j**: `Int`

`fn` next

SIGNATURE

```
next(mut self: Lag[N], var in_samp: SIMD[DType.float64, N], lag: SIMD[DType.float64, N] = 0.050000000000000003, num_lags: Int = $0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **in_samp**: `SIMD`
- **lag**: `SIMD` = `0.050000000000000003`
- **num_lags**: `Int` = `$0`

RETURNS

Type: `SIMD`

---### SVF

State Variable Filter implementation translated from Oleg Nesterov's Faust implementation

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.17 Functions

`fn` reset

Reset internal state

SIGNATURE

```
reset(mut self)
```

`fn` next

next a single sample through the SVF

SIGNATURE

```
next(mut self, input: SIMD[DType.float64, N], filter_type: SIMD[DType.int32, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain_db: SIMD[DType.float64, N] = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **filter_type**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`
- **gain_db**: `SIMD` = `0`

RETURNS

Type: `SIMD`

`fn` lpf

Lowpass filter

SIGNATURE

```
lpf(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`

RETURNS

Type: `SIMD`

`fn` bpf

Bandpass filter

SIGNATURE

```
bpf(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`

RETURNS

Type: `SIMD`

`fn` hpf

Highpass filter

SIGNATURE

```
hpf(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`

RETURNS

Type: `SIMD`

`fn` notch

Notch filter

SIGNATURE

```
notch(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`

RETURNS

Type: `SIMD`

`fn` peak

Peak filter

SIGNATURE

```
peak(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`

RETURNS

Type: `SIMD`

`fn` allpass

Allpass filter

SIGNATURE

```
allpass(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`

RETURNS

Type: `SIMD`

`fn` bell

Bell filter (parametric EQ)

SIGNATURE

```
bell(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain_db: SIMD[DType.float64, N]) -> SIMD[DType.flo
at64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`
- **gain_db**: `SIMD`

RETURNS

Type: `SIMD`

`fn` lowshelf

Low shelf filter

SIGNATURE

```
lowshelf(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain_db: SIMD[DType.float64, N]) ->
SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`
- **gain_db**: `SIMD`

RETURNS

Type: `SIMD`

`fn` highshelf

High shelf filter

SIGNATURE

```
highshelf(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain_db: SIMD[DType.float64, N]) -> SIMD[DTyp
e.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`
- **q**: `SIMD`
- **gain_db**: `SIMD`

RETURNS

Type: `SIMD`

---### lpf_LR4

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.18 Functions

`fn` set_sample_rate

SIGNATURE

```
set_sample_rate(mut self, sample_rate: Float64)
```

ARGUMENTS

- **sample_rate**: `Float64`

`fn` **next**

a single sample through the 4th order lowpass filter.

SIGNATURE

```
next(mut self, input: SIMD[DType.float64, N], frequency: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **frequency**: `SIMD`

RETURNS

Type: `SIMD`

---### OnePole

Simple one-pole IIR filter that can be configured as lowpass or highpass

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.19 Functions

`fn` **next**

Process one sample through the filter

SIGNATURE

```
next(mut self, input: Float64, coef: Float64) -> Float64
```

ARGUMENTS

- **input**: `Float64`
- **coef**: `Float64`

RETURNS

Type: `Float64`

---### Integrator

Simple one-pole IIR filter that can be configured as lowpass or highpass

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.20 Functions

`fn` **next**

Process one sample through the filter

SIGNATURE

```
next(mut self, input: Float64, coef: Float64) -> Float64
```

ARGUMENTS

- **input**: `Float64`
- **coef**: `Float64`

RETURNS

Type: `Float64`

---### DCTrap

DC Trap from Digital Sound Generation by Beat Frei.

Arguments: input: The input signal to process.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.21 Functions

`fn` **next**

Process one sample through the DC blocker filter

SIGNATURE

```
next(mut self, in_: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **in_**: `SIMD`

RETURNS

Type: `SIMD`

---### VAOnePole

Simple one-pole IIR filter that can be configured as lowpass or highpass}

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.22 Functions

`fn` **lpf**

Process one sample through the filter

SIGNATURE

```
lpf(mut self, input: SIMD[DType.float64, N], freq: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **freq**: `SIMD`

RETURNS

Type: `SIMD`

`fn` hpf

SIGNATURE

```
hpf(mut self, input: SIMD[DType.float64, N], freq: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **freq**: `SIMD`

RETURNS

Type: `SIMD`

---### VAMoogLadder

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.23 Functions

`fn` next

SIGNATURE

```
next(mut self, sig: SIMD[DType.float64, N], freq: SIMD[DType.float64, N], q_val: SIMD[DType.float64, N], os_index: Int = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **sig**: `SIMD`
- **freq**: `SIMD`
- **q_val**: `SIMD`
- **os_index**: `Int` = `0`

RETURNS

Type: `SIMD`

---### FIR

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.24 Functions

`fn` next

SIGNATURE

```
next(mut self: FIR[N], input: SIMD[DType.float64, N], coeffs: List[SIMD[DType.float64, N]]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **coeffs**: `List`

RETURNS

Type: `SIMD`

---### IIR

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.25 Functions

`fn` next

SIGNATURE

```
next(mut self: IIR[N], input: SIMD[DType.float64, N], coeffsbv: List[SIMD[DType.float64, N]], coeffsav: List[SIMD[DType.float64, N]]) -> SIMD[DType.float64,
N]
```

ARGUMENTS

- **input**: `SIMD`
- **coeffsbv**: `List`
- **coeffsav**: `List`

RETURNS

Type: `SIMD`

---### tf2

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.26 Functions

`fn` next

SIGNATURE

```
next(mut self: tf2[N], input: SIMD[DType.float64, N], coeffs: List[SIMD[DType.float64, N]]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **coeffs**: `List`

RETURNS

Type: `SIMD`

---### Reson

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.27 Functions

`fn` lpf

SIGNATURE

```
lpf(mut self: Reson[N], input: SIMD[DType.float64, N], freq: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain: SIMD[DType.float64, N]) -> SIMD[DType.fl
oat64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **freq**: `SIMD`
- **q**: `SIMD`
- **gain**: `SIMD`

RETURNS

Type: `SIMD`

`fn` hpf

SIGNATURE

```
hpf(mut self: Reson[N], input: SIMD[DType.float64, N], freq: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain: SIMD[DType.float64, N]) -> SIMD[DType.fl
oat64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **freq**: `SIMD`
- **q**: `SIMD`
- **gain**: `SIMD`

RETURNS

Type: `SIMD`

`fn` bpf

SIGNATURE

```
bpf(mut self: Reson[N], input: SIMD[DType.float64, N], freq: SIMD[DType.float64, N], q: SIMD[DType.float64, N], gain: SIMD[DType.float64, N]) -> SIMD[DType.fl
oat64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **freq**: `SIMD`
- **q**: `SIMD`
- **gain**: `SIMD`

RETURNS

Type: `SIMD`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.28 Env

Functions

MIN_ENV

Create a minimum envelope with specified ramp and duration.

Signature

```
min_env[N: Int = 1](ramp: SIMD[DType.float64, N] = 0.01, dur: SIMD[DType.float64, N] = 0.10000000000000001, rise: SIMD[DType.float64, N] = 0.001) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int`

Arguments

- **ramp**: `SIMD` = `0.01` - **dur**: `SIMD` = `0.10000000000000001` - **rise**: `SIMD` = `0.001`

Returns

Type: `SIMD`

ENV

Envelope generator.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.29 Functions

`fn` reset_vals

Reset internal values.

SIGNATURE

```
reset_vals(mut self, times: List[Float64])
```

ARGUMENTS

- **times**: `List`

`fn` next

Generate the next envelope sample.

SIGNATURE

```
next(mut self, values: List[Float64] = List[Float64, False](0, 1, 0, Tuple[]()), times: List[Float64] = List[Float64, False](1, 1, Tuple[]()), curves: List[Float64] = List[Float64, False](1, Tuple[]()), loop: Int64 = 0, trig: Float64 = 1, time_warp: Float64 = 1) -> Float64
```

ARGUMENTS

- **values**: `List` = `List[Float64, False](0, 1, 0, Tuple[]())`
- **times**: `List` = `List[Float64, False](1, 1, Tuple[]())`
- **curves**: `List` = `List[Float64, False](1, Tuple[]())`
- **loop**: `Int64` = `0`
- **trig**: `Float64` = `1`

- **time_warp**: `Float64` = `1`

RETURNS

Type: `Float64`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.30 Delays

DELAY

A delay line with Lagrange interpolation.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int` - size of the SIMD vector - defaults to 1

3.2.31 Functions

`fn` next

Process one sample through the delay line. This function computes the average of two values.next(input, delay_time)

SIGNATURE

```
next(mut self, input: SIMD[DType.float64, N], delay_time: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD` - The input sample to process.
- **delay_time**: `SIMD` - The amount of delay to apply (in seconds).

RETURNS

Type: `SIMD` The processed output sample.

`fn` lagrange4

Perform Lagrange interpolation for 4th order case (from JOS Faust Model).

SIGNATURE

```
lagrange4(mut self, input: SIMD[DType.float64, N], delay_time: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD`
- **delay_time**: `SIMD`

RETURNS

Type: `SIMD`

---### Comb

A simple comb filter using a delay line with feedback.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int` - size of the SIMD vector - defaults to 1

3.2.32 Functions

`fn` next

Process one sample through the comb filter.next(input, delay_time=0.0, feedback=0.0, interp=0)

SIGNATURE

```
next(mut self, input: SIMD[DType.float64, N], delay_time: SIMD[DType.float64, N] = 0, feedback: SIMD[DType.float64, N] = 0, interp: Int64 = 0) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD` - The input sample to process.
- **delay_time**: `SIMD` = `0` - The amount of delay to apply (in seconds).
- **feedback**: `SIMD` = `0` - The amount of feedback to apply (0.0 to 1.0).
- **interp**: `Int64` = `0` - The interpolation method to use (0 = linear, 1 = cubic, 2 = Lagrange).

RETURNS

Type: `SIMD` The processed output sample.

---### FBDelay

Like a Comb filter but with any amount of feedback and a tanh function.

```
Parameters:
  N: size of the SIMD vector - defaults to 1
```

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.33 Functions

`fn` next

Process one sample or SIMD vector through the feedback delay.next(input, delay_time=0.0, feedback=0.0, interp=0)

SIGNATURE

```
next(mut self, input: SIMD[DType.float64, N], delay_time: SIMD[DType.float64, N], feedback: SIMD[DType.float64, N]) -> SIMD[DType.float64, N]
```

ARGUMENTS

- **input**: `SIMD` - The input sample to process.
- **delay_time**: `SIMD` - The amount of delay to apply (in seconds).
- **feedback**: `SIMD` - The amount of feedback to apply (0.0 to 1.0).

RETURNS

Type: `SIMD` The processed output sample or SIMD vector.

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.34 Distortion

Functions

VTANH

Signature

```
vtanh(in_samp: Float64, gain: Float64, offset: Float64) -> Float64
```

Arguments

- **in_samp**: `Float64` - **gain**: `Float64` - **offset**: `Float64`

Returns

Type: `Float64`

BITCRUSHER

Signature

```
bitcrusher(in_samp: Float64, bits: Int64) -> Float64
```

Arguments

- **in_samp**: `Float64` - **bits**: `Int64`

Returns

Type: `Float64`

LATCH

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.35 Functions

`fn` next

SIGNATURE

```
next(mut self, in_samp: Float64, trig: Float64) -> Float64
```

ARGUMENTS

- **in_samp**: `Float64`
- **trig**: `Float64`

RETURNS

Type: `Float64`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.36 Noise

WHITENOISE

Generate white noise samples.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `UnknownDestructibility`

3.2.37 Functions

`fn` next

Generate the next white noise sample.

SIGNATURE

```
next(self, gain: Float64 = 1) -> Float64
```

ARGUMENTS

- **gain**: `Float64` = `1`

RETURNS

Type: `Float64` A random value between -gain and gain.

---### PinkNoise

Generate pink noise samples.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `UnknownDestructibility`

3.2.38 Functions

`fn` next

Generate the next pink noise sample.

SIGNATURE

```
next(mut self, gain: Float64 = 1) -> Float64
```

ARGUMENTS

- **gain**: `Float64` = `1` - Amplitude scaling factor.

RETURNS

Type: `Float64`

---### BrownNoise

Generate brown noise samples.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `UnknownDestructibility`

3.2.39 Functions

`fn` next

Generate the next brown noise sample.

SIGNATURE

```
next(mut self, gain: Float64 = 1) -> Float64
```

ARGUMENTS

- **gain**: `Float64` = `1` - Amplitude scaling factor.

RETURNS

Type: `Float64` A brown noise sample scaled by gain.

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.40 Pan

PAN2

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.41 Functions

`fn` **next**

SIGNATURE

```
next(mut self, sample: Float64, mut pan: Float64) -> SIMD[DType.float64, 2]
```

ARGUMENTS

- **sample**: `Float64`
- **pan**: `Float64`

RETURNS

Type: `SIMD`

---### PanAz

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.42 Functions

`fn` **next**

SIGNATURE

```
next[N: Int](mut self, sample: Float64, pan: Float64, num_speakers: Int64, width: Float64 = 2, orientation: Float64 = 0.5) -> SIMD[DType.float64, N]
```

PARAMETERS

- **N**: `Int`

ARGUMENTS

- **sample**: `Float64`
- **pan**: `Float64`
- **num_speakers**: `Int64`
- **width**: `Float64` = `2`
- **orientation**: `Float64` = `0.5`

RETURNS

Type: `SIMD`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.43 PlayBuf

PLAYBUF

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.44 Functions

`fn` next

get the next sample from an audio buffer - can take both Buffer or InterleavedBuffer.Arguments: buffer: The audio buffer to read from (can be Buffer or InterleavedBuffer). rate: The playback rate. 1 is the normal speed of the buffer. loop: Whether to loop the buffer (default: True). trig: Trigger starts the synth at start_frame (default: 1.0). start_frame: The start frame for playback (default: 0) upon receiving a trigger. end_frame: The end frame for playback (default: -1).

SIGNATURE

```
next[T: Buffable, N: Int = 1](mut self, mut buffer: T, start_chan: Int, rate: Float64, loop: Bool = True, trig: Float64 = 1, start_frame: Float64 = 0, end_frame: Float64 = -1) -> SIMD[DType.float64, N]
```

PARAMETERS

- **T**: `Buffable`
- **N**: `Int`

ARGUMENTS

- **buffer**: `T`
- **start_chan**: `Int`
- **rate**: `Float64`
- **loop**: `Bool` = `True`
- **trig**: `Float64` = `1`
- **start_frame**: `Float64` = `0`
- **end_frame**: `Float64` = `-1`

RETURNS

Type: `SIMD`

`fn` get_phase

SIGNATURE

```
get_phase(mut self) -> Float64
```

RETURNS

Type: `Float64`

`fn` get_win_phase

SIGNATURE

```
get_win_phase(mut self) -> Float64
```

RETURNS

Type: `Float64`

---### Grain

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.45 Functions

`fn` next

SIGNATURE

```
next[T: Buffable, N: Int = 1](mut self, mut buffer: T, start_chan: Int, trig: Float64 = 0, rate: Float64 = 1, start_frame: Float64 = 0, duration: Float64 = 0, pan: Float64 = 0, gain: Float64 = 1) -> SIMD[DType.float64, 2]
```

PARAMETERS

- **T**: `Buffable`
- **N**: `Int`

ARGUMENTS

- **buffer**: `T`
- **start_chan**: `Int`
- **trig**: `Float64` = `0`
- **rate**: `Float64` = `1`
- **start_frame**: `Float64` = `0`
- **duration**: `Float64` = `0`
- **pan**: `Float64` = `0`
- **gain**: `Float64` = `1`

RETURNS

Type: `SIMD`

---### TGrains

Triggered granular synthesis. Each trigger starts a new grain.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.46 Functions

`fn` next

Generate the next set of grains.Arguments:. buffer: Audio buffer containing the source sound. trig: Trigger signal (>0 to start a new grain). rate: Playback rate of the grains (1.0 = normal speed). start_frame: Starting frame position in the buffer. duration: Duration of each grain in seconds. pan: Panning position from -1.0 (left) to 1.0 (right). gain: Amplitude scaling factor for the grains.

SIGNATURE

```
next[T: Buffable, N: Int = 1](mut self, mut buffer: T, buf_chan: Int, trig: Float64 = 0, rate: Float64 = 1, start_frame: Float64 = 0, duration: Float64 = 0.10000000000000001, pan: Float64 = 0, gain: Float64 = 1) -> SIMD[DType.float64, 2]
```

PARAMETERS

- **T**: `Buffable`
- **N**: `Int`

ARGUMENTS

- **buffer**: `T`

- **buf_chan**: `Int`
- **trig**: `Float64` = `0`
- **rate**: `Float64` = `1`
- **start_frame**: `Float64` = `0`
- **duration**: `Float64` = `0.10000000000000001`
- **pan**: `Float64` = `0`
- **gain**: `Float64` = `1`

RETURNS

Type: `SIMD` List of output samples for all channels.

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.47 RecordBuf

RECORDBUF

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.48 Functions

`fn` **write**

SIGNATURE

```
write[N: Int = 1](mut self, value: SIMD[DType.float64, N], mut buffer: Buffer)
```

PARAMETERS

- **N**: `Int`

ARGUMENTS

- **value**: `SIMD`
- **buffer**: `Buffer`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.49 Oversampling

OVERSAMPLING

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

Parameters

1. **N**: `Int`

3.2.50 Functions

`fn` set_os_index

SIGNATURE

```
set_os_index(mut self, index: Int)
```

ARGUMENTS

- **index**: `Int`

`fn` add_sample

Add a sample to the oversampling buffer.

SIGNATURE

```
add_sample(mut self, sample: SIMD[DType.float64, N])
```

ARGUMENTS

- **sample**: `SIMD`

`fn` get_sample

get the next sample from a filled oversampling buffer.

SIGNATURE

```
get_sample(mut self) -> SIMD[DType.float64, N]
```

RETURNS

Type: `SIMD`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.2.51 MLP

MLP

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.2.52 Functions

`fn` next

SIGNATURE

```
next[N: Int = 16](mut self, input: List[Float64]) -> SIMD[DType.float64, N]
```

PARAMETERS

- **N**: `Int`

ARGUMENTS

- **input**: `List`

RETURNS

Type: `SIMD`

RAISES

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.3 Core Framework

3.3.1 MMMGraphs

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.3.2 MMMTraits

Traits

GRAPHABLE

Signature

```
```

Required Methods

next Signature

```
next(mut self: _Self) -> List[Float64]
```

Arguments

- **self**: `_Self`

Returns

Type: `List`

BUFFABLE

Signature

```
```

Required Methods

read Signature

```
read[N: Int = 1](mut self: _Self, start_chan: Int64, phase: Float64, interp: Int64 = 0) -> SIMD[DType.float64, N]
```

Parameters

- **N**: `Int`

Arguments

- **self**: `_Self` - **start_chan**: `Int64` - **phase**: `Float64` - **interp**: `Int64` = `0`

Returns

Type: `SIMD`

---

get_num_frames Signature

```
get_num_frames(self: _Self) -> Float64
```

Arguments

- **self**: `_Self`

Returns

Type: `Float64`

---

get_duration Signature

```
get_duration(self: _Self) -> Float64
```

Arguments

- **self**: `_Self`

Returns

Type: `Float64`

---

get_buf_sample_rate Signature

```
get_buf_sample_rate(self: _Self) -> Float64
```

Arguments

- **self**: `_Self`

Returns

Type: `Float64`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.3.3 MMMWorld

MMMWORLD

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.3.4 Functions

`fn` set_channel_count

SIGNATURE

```
set_channel_count(mut self, num_in_chans: Int64, num_out_chans: Int64)
```

ARGUMENTS

- **num_in_chans**: `Int64`
- **num_out_chans**: `Int64`

`fn` send_msg

SIGNATURE

```
send_msg(mut self, key: String, mut list: List[Float64])
```

ARGUMENTS

- **key**: `String`
- **list**: `List`

`fn` get_msg

SIGNATURE

```
get_msg(mut self, key: String) -> Optional[List[Float64]]
```

ARGUMENTS

- **key**: `String`

RETURNS

Type: `Optional`

`fn` send_text_msg

SIGNATURE

```
send_text_msg(mut self, key: String, mut list: List[String])
```

ARGUMENTS

- **key**: `String`
- **list**: `List`

`fn` get_text_msg

SIGNATURE

```
get_text_msg(mut self, key: String) -> Optional[List[String]]
```

ARGUMENTS

- **key**: `String`

RETURNS

Type: `Optional`

`fn` get_midi

SIGNATURE

```
get_midi(mut self, key: String, chan: Int64 = -1, param: Int64 = -1) -> Optional[List[List[Int64]]]
```

ARGUMENTS

- **key**: `String`
- **chan**: `Int64` = `-1`
- **param**: `Int64` = `-1`

RETURNS

Type: `Optional`

`fn` clear_midi

SIGNATURE

```
clear_midi(mut self)
```

`fn` send_midi

SIGNATURE

```
send_midi(mut self, msg: PythonObject)
```

ARGUMENTS

- **msg**: `PythonObject`

RAISES

`fn` clear_msgs

SIGNATURE

```
clear_msgs(mut self)
```

`fn` print

SIGNATURE

```
print[T: Writable](mut self, value: T, label: String = "", freq: Float64 = 10)
```

PARAMETERS

- **T**: `Writable`

ARGUMENTS

- **value**: `T`
- **label**: `String` = `""`
- **freq**: `Float64` = `10`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.4 Utilities

3.4.1 MMM_FFT

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.4.2 Windows

Functions

BESSEL_I0

Calculate the modified Bessel function of the first kind, order 0 (I_0). Uses polynomial approximation for accurate results.

Signature

```
bessel_i0(x: Float64) -> Float64
```

Arguments

- **x**: `Float64` - Input value

Returns

Type: `Float64`

KAISER_WINDOW

Create a Kaiser window of length n with shape parameter beta.

Signature

```
kaiser_window(size: Int64, beta: Float64) -> List[Float64]
```

Arguments

- **size**: `Int64` - **beta**: `Float64` - Shape parameter that controls the trade-off between main lobe width and side lobe level
- beta = 0: rectangular window
- beta = 5: similar to Hamming window
- beta = 6: similar to Hanning window
- beta = 8.6: similar to Blackman window

Returns

Type: `List` DynamicVector[Float64] containing the Kaiser window coefficients

---

BUILD_SINC_TABLE

Build a sinc function lookup table.

Signature

```
build_sinc_table(size: Int64, ripples: Int64 = 4) -> List[Float64]
```

Arguments

- **size**: `Int64` - Number of points in the table- **ripples**: `Int64` = `4` - Number of ripples/lobes on each side of the main lobe

Returns

Type: `List` List containing the sinc function values

HANN_WINDOW

Generate a Hann window of length n.

Signature

```
hann_window(n: Int64) -> List[Float64]
```

Arguments

- **n**: `Int64` - Length of the window

Returns

Type: `List`

HAMMING_WINDOW

Generate a Hamming window of length n.

Signature

```
hamming_window(n: Int64) -> List[Float64]
```

Arguments

- **n**: `Int64`

Returns

Type: `List`

BLACKMAN_WINDOW

Generate a Blackman window of length n. Args: n: Length of the window Returns: List containing the Blackman window values

Signature

```
blackman_window(n: Int64) -> List[Float64]
```

Arguments

- **n**: `Int64`

Returns

Type: `List`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

3.4.3 Print

PRINT

A struct for printing values in the MMMWorld environment.

Parent Traits: `AnyType`, `Copyable`, `Movable`, `Representable`, `UnknownDestructibility`

3.4.4 Functions

`fn` next

Print the value at a given frequency.Arguments: value: The value to print. label: An optional label to prepend to the printed value. freq: The frequency (in Hz) at which to print the value.

SIGNATURE

```
next[T: Writable](mut self, value: T, label: String = "", freq: Float64 = 10)
```

PARAMETERS

- **T**: `Writable`

ARGUMENTS

- **value**: `T`
- **label**: `String` = `""`
- **freq**: `Float64` = `10`

Documentation generated with `mojo doc' from Mojo version 0.25.6.0.dev2025090605

4. Examples

4.1 Examples

This section contains practical examples demonstrating how to use MMMAudio for various audio processing tasks.

4.1.1 Basic Examples

- **Default Graph**: Basic audio graph setup
- **In2Out**: Simple input to output routing

4.1.2 Synthesis Examples

- **Many Oscillators**: Multiple oscillator management
- **Grains**: Granular synthesis techniques

4.1.3 Effects Examples

- **Feedback Delays**: Delay-based effects
- **Pan Az**: Spatial audio panning

4.1.4 Advanced Examples

- **MIDI Sequencer**: MIDI-controlled sequencing
- **Torch MLP**: Neural network audio processing
- **Record**: Audio recording and playback

4.1.5 Running Examples

Most examples can be run directly with Python:

```
python examples/default.py
```

Or with Mojo for the .mojo examples:

```
mojo examples/Default_Graph.mojo
```

5. Contributing

5.1 Overview

test

5.2 Documentation Guide

This guide explains how to write and maintain documentation for MMMAudio.

5.2.1 Documentation Structure

MMMAudio uses MkDocs with Material theme for documentation generation. The documentation supports both Python and Mojo source files through different processing pipelines.

Python Documentation

Python files are documented using Google-style docstrings and processed by mkdocstrings:

```
def example_function(param1: int, param2: str = "default") -> bool:
    """Brief description of the function.

    Longer description with more details about what the function does,
    its intended use cases, and any important behavior notes.

    Args:
        param1: Description of the first parameter.
        param2: Description of the second parameter with default value.

    Returns:
        Description of the return value.

    Raises:
        ValueError: When param1 is negative.

    Examples:
        Basic usage:

        ```python
        result = example_function(42, "test")
        ```

        With default parameter:

        ```python
        result = example_function(42)
        ```
    """
    return param1 > 0
```

Mojo Documentation

Mojo files are documented using triple-quoted docstrings and processed by our custom adapter:

```
fn example_function[N: Int = 1](
    param1: SIMD[DType.int32, N],
    param2: SIMD[DType.float64, N] = 1.0
) -> SIMD[DType.bool, N]:
    """Brief description of the function.

    Longer description with details about the function's behavior,
    SIMD optimization, and usage patterns.

    Parameters:
        N: SIMD vector width (defaults to 1).

    Args:
        param1: Description of the first parameter.
        param2: Description of the second parameter with default.

    Returns:
        Description of the return value.

    Examples:
        Single value processing:

        ```mojo
        result = example_function(42, 1.5)
        ```

        Vectorized processing:

        ```mojo
        values = SIMD[DType.int32, 4](1, 2, 3, 4)
        factors = SIMD[DType.float64, 4](1.0, 1.5, 2.0, 2.5)
```
```

```
        results = example_function[4](values, factors)
        ```
    """
    return param1 > 0
```

## 5.2.2 Building Documentation

### Prerequisites

Install documentation dependencies:

```
pip install -r requirements-docs.txt
```

Generate Documentation

Run the documentation pipeline:

```
python documentation/generate_docs.py
```

This will:
1. Process all Mojo files and extract documentation
2. Create markdown stubs for Python files
3. Generate example documentation
4. Build the complete documentation site

Serve Locally

To preview the documentation locally:

```
mkdocs serve
```

The documentation will be available at `http://localhost:8000` .

Build for Production

To build the static documentation site:

```
mkdocs build
```

This creates a `site/` directory with the complete documentation.

Build PDF

To generate a PDF version:

```
mkdocs build
# PDF is generated automatically by mkdocs-pdf plugin
```

5.2.3 Documentation Standards

Writing Guidelines

1. **Be Clear and Concise**: Use simple, direct language
2. **Include Examples**: Every function should have practical examples
3. **Explain SIMD Usage**: For Mojo functions, explain vectorization benefits
4. **Cross-Reference**: Link to related functions and concepts
5. **Keep Updated**: Update docs when code changes
```

Code Examples

- Use real, runnable code examples
- Show both basic and advanced usage
- Include expected output when helpful
- Demonstrate error conditions when relevant

Function Documentation

Required sections: - Brief description (first line) - Detailed description - Parameters/Args with types and descriptions - Returns section - Examples section

Optional sections: - Raises (for error conditions) - Notes (for implementation details) - See Also (for related functions)

5.2.4 Maintenance

Regular Updates

- Review documentation when adding new features
- Update examples to use current best practices
- Check for broken links and outdated information
- Ensure all public APIs are documented

Documentation Reviews

Include documentation updates in code reviews: - Verify new functions are documented - Check that examples are correct and clear - Ensure docstring formatting is consistent - Validate that generated docs look correct

5.3 Contributing to MMMAudio

Thank you for your interest in contributing to MMMAudio! This guide will help you get started.

5.3.1 Development Setup

Prerequisites

- Python 3.9+
- Mojo compiler (latest version)
- Git

Installation

1. Fork and clone the repository:

```
git clone https://github.com/your-username/MMMAudio.git
cd MMMAudio
```

1. Install dependencies:

```
pip install -r requirements-docs.txt
```

1. Verify the setup:

```
python -c "import mmm_src.MMMAudio; print('Python setup OK')"
mojo --version
```

5.3.2 Contributing Guidelines

Code Style

PYTHON CODE

- Follow PEP 8 style guidelines
- Use type hints for all function signatures
- Use Google-style docstrings
- Maximum line length: 88 characters (Black formatter)

MOJO CODE

- Follow Mojo style conventions
- Use SIMD types for performance-critical code
- Document all public functions with examples
- Use descriptive variable names

Documentation

- All public APIs must be documented
- Include practical examples for each function
- Update documentation when changing functionality
- Use the documentation templates in `documentation/`

Testing

- Add tests for new functionality
- Ensure existing tests pass
- Test both Python and Mojo implementations
- Include performance benchmarks for critical paths

Pull Request Process

1. Create a feature branch:

```
git checkout -b feature/your-feature-name
```

2. Make your changes with clear, atomic commits
3. Update documentation and tests
4. Generate and review documentation:

```
python documentation/generate_docs.py
mkdocs serve
```

5. Submit a pull request with:
6. Clear description of changes
7. Rationale for the changes
8. Any breaking changes noted
9. Screenshots of documentation updates

Issue Reporting

When reporting issues: - Use the issue templates - Include minimal reproduction case - Specify OS and version information - Include relevant error messages and logs

5.3.3 Development Workflow

Adding New DSP Functions

1. Implement in Mojo for performance (in `mmm_dsp/`)
2. Add comprehensive documentation with examples
3. Create Python wrapper if needed (in `mmm_src/`)
4. Add tests demonstrating functionality
5. Update relevant examples

Adding New Examples

1. Create example in `examples/` directory
2. Include clear documentation in docstring
3. Ensure example runs without errors
4. Add to examples index in documentation

Performance Optimization

- Profile before optimizing
- Use SIMD operations where possible
- Benchmark against reference implementations

- Document performance characteristics

5.3.4 Community

Communication

- GitHub Issues: Bug reports and feature requests
- Discussions: General questions and community interaction
- Pull Requests: Code contributions and reviews

Code of Conduct

Please be respectful and constructive in all interactions. We're building a welcoming community for audio developers of all skill levels.

5.3.5 Release Process

Releases follow semantic versioning: - Major: Breaking changes - Minor: New features (backward compatible) - Patch: Bug fixes

Documentation is automatically built and deployed with each release.