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

```
```

1.3 Documentation Structure

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

1.5.1 Why Python?

1.5.2 Why Mojo?

1.6 Community

- **GitHub**: https://github.com/spluta/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], sample4: 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

Envelope generator module.

This module provides an envelope generator class that can create complex envelopes with multiple segments, curves, and looping capabilities.

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.f
loat64, 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
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the [Examples Overview]

4.2 Default_Graph

use this as a template for your own graphs

4.2.1 Python Code

```
from mmm_src.MMMAudio import MMMAudio


# instantiate and load the graph
mmm_audio = MMMAudio(128, graph_name="Default_Graph", package_name="examples")
mmm_audio.start_audio()

from random import random
mmm_audio.send_msg("osc_freq", random() * 500 + 100 ) # set the frequency to a random value
```

4.2.2 Mojo Code

```
"""use this as a template for your own graphs"""

from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *

from mmm_dsp.Osc import Osc
from mmm_dsp.Filters import Lag

struct Default_Synth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var osc: Osc
    var freq: Float64
    var lag: Lag

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.osc = Osc(self.world_ptr)
        self.freq = 440.0
        self.lag = Lag(self.world_ptr)

    fn __repr__(self) -> String:
        return String("Default")

    fn next(mut self) -> Float64:

        self.get_msgs()
        freq = self.lag.next(self.freq, 3)
        return self.osc.next(freq[0]) * 0.1

    fn get_msgs(mut self: Self):
        # Get messages from the world
        msg = self.world_ptr[0].get_msg("osc_freq")
        if msg:
            self.freq = msg.value()[0]

# there can only be one graph in an MMMAudio instance
# a graph can have as many synths as you want
struct Default_Graph(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var synth: Default_Synth

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.synth = Default_Synth(self.world_ptr)

    fn __repr__(self) -> String:
        return String("Default_Graph")

    fn next(mut self) -> SIMD[DType.float64, 1]:

        return self.synth.next()  # Get the next sample from the synth
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the [Examples Overview](#)

4.3 In2Out

This is the simples MMMAudio example. It routes input channels directly to output channels. It also demonstrates how to send a message to the graph to print the current input values to the REPL.

4.3.1 Python Code

```python
from mmm_src.MMMAudio import *

# this will list available audio devices
list_audio_devices()

in_device = "Fireface UFX+ (24059506)"
out_device = "Fireface UFX+ (24059506)"

# or get some feedback
in_device = "MacBook Pro Microphone"
out_device = "External Headphones"

# instantiate and load the graph
mmm_audio = MMMAudio(128, num_input_channels=12, num_output_channels=12, in_device=in_device, out_device=out_device, graph_name="In2Out", package_name="exampl
es")
mmm_audio.start_audio()

# print the current sample of inputs to the REPL
mmm_audio.send_msg("print_inputs")

mmm_audio.stop_audio()
```

4.3.2 Mojo Code

```mojo
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *


# this is the simplest possible
struct In2Out(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr

    fn __repr__(self) -> String:
        return String("In2Out")

    fn next(mut self) -> SIMD[DType.float64, 16]:
        self.get_msgs()

        # the SIMD vector has to be a power of 2
        output = SIMD[DType.float64, 16](0.0)

        # whichever is smaller, the output or the sound_in - that number of values are copied to the output
        smaller  = min(len(output), len(self.world_ptr[0].sound_in))
        for i in range(smaller):
            output[i] = self.world_ptr[0].sound_in[i]

        return output  # Return the combined output samples

    fn get_msgs(mut self: Self):
        # a "print_inputs" message prints the current values held in the sound_in list in the world_ptr
        msg = self.world_ptr[0].get_msg("print_inputs")
        if msg:
            for i in range(self.world_ptr[0].num_in_chans):
                print("input[", i, "] =", self.world_ptr[0].sound_in[i])
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.4 Pan_Az

4.4.1 Python Code

```
from mmm_src.MMMAudio import MMMAudio


# instantiate and load the graph

# PanAz is not quite right as of yet
mmm_audio = MMMAudio(128, graph_name="Pan_Az", package_name="examples", num_output_channels=5)
mmm_audio.start_audio()

mmm_audio.stop_audio()

from random import random
mmm_audio.send_msg("osc_freq", random() * 500 + 100 ) # set the frequency to a random value
```

4.4.2 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *

from mmm_dsp.Osc import Phasor, Osc
from mmm_dsp.Pan import PanAz

struct PanAz_Synth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var osc: Osc
    var freq: Float64

    var pan_osc: Phasor
    var pan_az: PanAz

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.osc = Osc(self.world_ptr)
        self.freq = 440.0

        self.pan_osc = Phasor(self.world_ptr)
        self.pan_az = PanAz(self.world_ptr)

    fn __repr__(self) -> String:
        return String("Default")

    fn next(mut self) -> SIMD[DType.float64, 8]:

        self.get_msgs()

        # PanAz needs to be given a SIMD size that is a power of 2, in this case [8], but the speaker size can be anything smaller than that
        panned = self.pan_az.next[8](self.osc.next(self.freq, osc_type=2), self.pan_osc.next(0.1), 2, 2) * 0.1

        return panned

    fn get_msgs(mut self: Self):
        # Get messages from the world
        msg = self.world_ptr[0].get_msg("osc_freq")
        if msg:
            self.freq = msg.value()[0]

# there can only be one graph in an MMMAudio instance
# a graph can have as many synths as you want
struct Pan_Az(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var synth: PanAz_Synth

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.synth = PanAz_Synth(self.world_ptr)

    fn __repr__(self) -> String:
        return String("PanAz")

    fn next(mut self) -> SIMD[DType.float64, 8]:

        sample = self.synth.next()  # Get the next sample from the synth

        # the output will pan to the number of channels available
```

```
        # if there are fewer than 5 channels, only those channels will be output
        return sample  # Return the combined output samples
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.5 ManyOscillators

Example showing how to use ManyOscillators.mojo with MMMAudio.

You can change the number of oscillators dynamically by sending a 'set_num_pairs' message.

4.5.1 Python Code

```
from mmm_src.MMMAudio import MMMAudio

mmm_audio = MMMAudio(128, graph_name="ManyOscillators", package_name="examples")

mmm_audio.start_audio() # start the audio thread - or restart it where it left off

mmm_audio.send_msg("set_num_pairs", 2)  # set to 2 pairs of oscillators

mmm_audio.send_msg("set_num_pairs", 14)  # change to 4 pairs of oscillators

mmm_audio.send_msg("set_num_pairs", 50)  # change to 4 pairs of oscillators

mmm_audio.stop_audio() # stop/pause the audio thread
```

4.5.2 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *

from mmm_dsp.Osc import Osc
from random import random_float64
from mmm_dsp.Pan import Pan2
from mmm_dsp.OscBuffers import OscBuffers

# THE SYNTH

struct OscSynth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var oscs: Osc[2]  # An Osc instance with two internal Oscs
    var osc_freqs: SIMD[DType.float64, 2]
    var pan: Pan2
    var pan_osc: Osc
    var pan_freq: Float64
    var vol_osc: Osc
    var vol_osc_freq: Float64
    var temp: Float64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld], center_freq: Float64):
        self.world_ptr = world_ptr
        self.oscs = Osc[2](world_ptr)  # Initialize two Osc instances

        self.pan = Pan2(world_ptr)
        self.pan_osc = Osc(world_ptr)
        self.pan_freq = random_float64(0.03, 0.1)

        self.vol_osc = Osc(world_ptr)
        self.vol_osc_freq = random_float64(0.05, 0.2)
        self.osc_freqs = SIMD[DType.float64, 2](
            center_freq + random_float64(1.0, 5.0),
            center_freq - random_float64(1.0, 5.0)
        )
        self.temp = 0.0

    fn __repr__(self) -> String:
        return String("OscSynth")

    fn next(mut self) -> SIMD[DType.float64, 2]:

        temp = self.oscs.next(self.osc_freqs, interp = 0, os_index = 0)

        temp = temp * (self.vol_osc.next(self.vol_osc_freq) * 0.01 + 0.01)
        temp2 = temp[0] + temp[1]

        self.world_ptr[0].print(self.osc_freqs, "freqs", freq=1.0)

        pan_loc = self.pan_osc.next(self.pan_freq)  # Get pan position

        return self.pan.next(temp2, pan_loc)  # Pan the temp signal

# THE GRAPH
```

```
struct ManyOscillators(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]

    var osc_synths: List[OscSynth]  # Instances of the Oscillator
    var num_pairs: Int

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr

        # initialize the list of oscillator pairs
        self.osc_synths = List[OscSynth]()
        # add 10 pairs to the list
        self.num_pairs = 10
        for _ in range(self.num_pairs):
            self.osc_synths.append(OscSynth(self.world_ptr, random_exp_float64(100.0, 1000.0)))

    fn __repr__(self) -> String:
        return String("ManyOscillators")

    fn next(mut self) -> SIMD[DType.float64, 2]:
        self.get_msgs()

        # sum all the stereo outs from the N oscillator pairs
        sum = SIMD[DType.float64, 2](0.0, 0.0)
        for i in range(self.num_pairs):
            sum += self.osc_synths[i].next()

        return sum

    fn get_msgs(mut self):
        # looking for a message that changes the number of osc pairs

        num = self.world_ptr[0].get_msg("set_num_pairs")
        if num:
            if num.value()[0] != self.num_pairs:
                print("Changing number of osc pairs to:", Int(num.value()[0]))
                # adjust the list of osc synths
                if Int(num.value()[0]) > self.num_pairs:
                    # add more
                    for _ in range(Int(num.value()[0]) - self.num_pairs):
                        self.osc_synths.append(OscSynth(self.world_ptr, random_exp_float64(100.0, 1000.0)))
                else:
                    # remove some
                    for _ in range(self.num_pairs - Int(num.value()[0])):
                        _ = self.osc_synths.pop()
            self.num_pairs = Int(num.value()[0])
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.6 FeedbackDelays

use the mouse to control an overdriven feedback delay

4.6.1 Python Code

```
from mmm_src.MMMAudio import MMMAudio

mmm_audio = MMMAudio(128, graph_name="FeedbackDelays", package_name="examples")

mmm_audio.start_audio() # start the audio thread - or restart it where it left off
mmm_audio.stop_audio() # stop/pause the audio thread
```

4.6.2 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *

from mmm_dsp.Buffer import *
from mmm_dsp.PlayBuf import *
from mmm_dsp.Delays import *
from mmm_utils.functions import *

struct DelaySynth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]

    var buffer: InterleavedBuffer  # Interleaved buffer for audio samples
    var playBuf: PlayBuf
    var delays: FBDelay[2]  # FBDelay for feedback delay effect
    var lag: Lag[2]
    var mouse_x: Float64
    var mouse_y: Float64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.buffer = InterleavedBuffer(self.world_ptr, "resources/Shiverer.wav")
        self.playBuf = PlayBuf(self.world_ptr)
        # FBDelay is initialized as 2 channel
        self.delays = FBDelay[2](self.world_ptr)

        self.lag = Lag[2](self.world_ptr)  # Initialize Lag with a default time constant

        self.mouse_x = 0.0
        self.mouse_y = 0.0

    fn next(mut self) -> SIMD[DType.float64, 2]:
        self.get_msgs()  # Get messages from the world
        var sample = self.playBuf.next[N=2](self.buffer, 0, 1.0, True)  # Read samples from the buffer

        # sending one value to the 2 channel lag gives both lags the same parameters
        # var del_time = self.lag.next(linlin(self.mouse_x, 0.0, 1.0, 0.0, self.buffer.get_duration()), 0.5)

        # this is a version with the 2 value SIMD vector as input each delay with have its own del_time
        var del_time = self.lag.next(SIMD[DType.float64, 2](
            linlin(self.mouse_x, 0.0, 1.0, 0.0, self.buffer.get_duration()),
            linlin(self.mouse_x, 0.0, 1.0, 0.0, self.buffer.get_duration()*0.9)
        ), SIMD[DType.float64, 2](0.5, 0.5))

        var feedback = SIMD[DType.float64, 2](self.mouse_y * 2.0, self.mouse_y * 2.1)

        sample = self.delays.next(sample, del_time, feedback)*0.8

        return sample

    fn __repr__(self) -> String:
        return String("DelaySynth")

    fn get_msgs(mut self):
        self.mouse_x = self.world_ptr[0].mouse_x
        self.mouse_y = self.world_ptr[0].mouse_y

struct FeedbackDelays(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var delay_synth: DelaySynth  # Instance of the Oscillator

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.delay_synth = DelaySynth(world_ptr)  # Initialize the DelaySynth with the world instance
```

```
    fn __repr__(self) -> String:
        return String("FeedbackDelays")

    fn next(mut self: FeedbackDelays) -> SIMD[DType.float64, 2]:
        return self.delay_synth.next()  # Return the combined output sample
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the [Examples Overview]

4.7 Grains

this uses the mouse to control granular playback of the buffer left and right moves around in the buffer. up and down controls rate of triggers.

4.7.1 Python Code

```
from mmm_src.MMMAudio import MMMAudio

mmm_audio = MMMAudio(128, graph_name="Grains", package_name="examples")

mmm_audio.start_audio() # start the audio thread - or restart it where it left off
mmm_audio.stop_audio() # stop/pause the audio thread
```

4.7.2 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *

from mmm_dsp.Buffer import *
from mmm_dsp.PlayBuf import *
from mmm_dsp.Osc import *
from mmm_dsp.Filters import VAMoogLadder
from mmm_utils.functions import linexp
from random import random_float64

# THE SYNTH

struct GrainSynth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var buffer: InterleavedBuffer

    var num_chans: Int64

    var tgrains: TGrains
    var impulse: Impulse
    var start_frame: Float64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr

        # interleaved buffer uses numpy to load a buffer into an interleaved array
        self.buffer = InterleavedBuffer(self.world_ptr, "resources/Shiverer.wav")
        self.num_chans = self.buffer.num_chans

        # it will try to free the interleaved buffer if you don't print here. gotta figure this out. this is either a bug by me or by modular.
        print("Loaded buffer with ", self.buffer.get_num_frames(), " frames and ", self.num_chans, " channels.")

        self.tgrains = TGrains(self.world_ptr, 20)
        self.impulse = Impulse(self.world_ptr)


        self.start_frame = 0.0

    fn next(mut self) -> SIMD[DType.float64, 2]:

        imp_freq = linlin(self.world_ptr[0].mouse_y, 0.0, 1.0, 5.0, 40.0)
        var impulse = self.impulse.next(imp_freq, 1.0)  # Get the next impulse sample

        start_frame = linlin(self.world_ptr[0].mouse_x, 0.0, 1.0, 0.0, self.buffer.get_num_frames())

        # use the first channel of the buffer
        var grains = self.tgrains.next(self.buffer, 0, impulse, 1, start_frame, 0.4, random_float64(-1.0, 1.0), 0.4)

        # if you want to use both channels of the buffer, uncomment this and comment the line above
        # with the 2 channel version, there will be 2 channels of output (in stereo), but no panning
        # var grains = self.tgrains.next[N=2](self.buffer, 0, impulse, 1, start_frame, 0.4, random_float64(-1.0, 1.0), 0.4)

        return grains


    fn __repr__(self) -> String:
        return String("GrainSynth")

# THE GRAPH

struct Grains(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var grain_synth: GrainSynth  # Instance of the GrainSynth
```

```
    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr

        self.grain_synth = GrainSynth(world_ptr)  # Initialize the GrainSynth with the world instance

    fn __repr__(self) -> String:
        return String("TGrains")

    fn next(mut self: Grains) -> SIMD[DType.float64, 2]:
        sample = self.grain_synth.next()

        return sample  # Return the combined output sample
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.8 Record

4.8.1 Python Code

```
from mmm_src.MMMAudio import *
list_audio_devices()

in_device = "Fireface UFX+ (24059506)"
out_device = "Fireface UFX+ (24059506)"

in_device = "MacBook Pro Microphone"
out_device = "External Headphones"



# instantiate and load the graph
mmm_audio = MMMAudio(128, num_input_channels=12, num_output_channels=12, in_device=in_device, out_device=out_device, graph_name="Record", package_name="exampl
es")

# the default input channel (in the Record_Synth) is 0, but you can change it to
mmm_audio.send_msg("set_input_chan", 0)
mmm_audio.start_audio()



mmm_audio.send_msg("print_inputs")

# this program is looking for midi note_on and note_off from note 48, so we prepare the keyboard to send messages to mmm_audio:

import mido
import time
import threading

# find your midi devices
mido.get_input_names()

# open your midi device - you may need to change the device name
in_port = mido.open_input('Oxygen Pro Mini USB MIDI')

def start_midi():
    while True:
        for msg in in_port.iter_pending():
            print(msg)
            mmm_audio.send_midi(msg)
        time.sleep(0.01) # Small delay to prevent busy-waiting

midi_thread = threading.Thread(target=start_midi, daemon=True)
# once you start the midi_thread, it should register note_on, note_off, cc, etc from your device and send them to mmm
midi_thread.start()


midi_thread.stop()

mmm_audio.stop_audio()
```

4.8.2 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *
from mmm_dsp.Buffer import *
from mmm_dsp.RecordBuf import RecordBuf
from mmm_dsp.PlayBuf import PlayBuf
from mmm_dsp.Env import min_env
from mmm_dsp.Filters import Lag

import time
from math import floor

from mmm_dsp.Osc import Osc

struct Record_Synth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var buf_dur: Float64
    var buffer: Buffer
    var is_recording: Float64
    var is_playing: Float64
    var playback_speed: Float64
    var trig: Float64
    var write_pos: Int64
    var record_buf: RecordBuf
```

```
    var play_buf: PlayBuf
    var note_ons: List[List[Int64]]
    var note_offs: List[List[Int64]]
    var note_time: Float64
    var lag: Lag
    var end_frame: Float64
    var input_chan: Int64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.buf_dur = 10.0  # seconds
        self.buffer = Buffer(1, Int64(self.world_ptr[0].sample_rate*self.buf_dur), self.world_ptr[0].sample_rate)
        self.is_recording = 0.0
        self.is_playing = 0.0
        self.trig = 0.0
        self.playback_speed = 1.0
        self.record_buf = RecordBuf(world_ptr)
        self.play_buf = PlayBuf(world_ptr)
        self.write_pos = 0
        self.note_ons = List[List[Int64]]()
        self.note_offs = List[List[Int64]]()
        self.note_time = 0.0
        self.end_frame = 0.0
        self.lag = Lag(world_ptr)
        self.input_chan = 0

    fn __repr__(self) -> String:
        return String("Record_Synth")

    fn next(mut self) -> SIMD[DType.float64, 1]:
        self.get_msgs()

        for note_on in self.note_ons:
            print(note_on[0], note_on[1], note_on[2], end = "\n")
            if note_on[1] == 48:
                self.note_time = time.perf_counter()
                self.write_pos = 0
                self.is_recording = 1.0
                self.is_playing = 0.0
                self.trig = 0.0
                print("Recording started")
        self.note_ons.clear()

        for note_off in self.note_offs:
            if note_off[1] == 48:  # C4 to stop recording
                self.note_time = min(time.perf_counter() - self.note_time, self.buf_dur)
                self.end_frame = floor(self.note_time*self.world_ptr[0].sample_rate)
                print(self.note_time, self.end_frame/self.world_ptr[0].sample_rate)
                self.note_time = self.end_frame / self.world_ptr[0].sample_rate
                print("Recorded duration:", self.note_time, "seconds")
                self.is_recording = 0.0
                print("Recording stopped. Now playing.")
                self.is_playing = 1.0
                self.trig = 1.0
                self.write_pos = 0

        self.note_offs.clear()

        # this code does the actual recording, placing the next sample into the buffer
        # my audio interface has audio in on channel 9, so I use self.world_ptr[0].sound_in[8]
        if self.is_recording:
            # the sound_in List in the world_ptr holds the audio in data for the current sample, so grab it from there.
            self.buffer.write(self.world_ptr[0].sound_in[self.input_chan], self.write_pos)
            self.write_pos += 1
            if self.write_pos >= Int(self.buffer.num_frames):
                self.is_recording = 0.0
                print("Recording stopped: buffer full")
                self.is_playing = 1.0
                self.trig = 1.0
                self.write_pos = 0

        out = self.play_buf.next(self.buffer, 0, self.playback_speed, True, self.trig, start_frame = 0, end_frame = self.end_frame)

        self.world_ptr[0].print(String(self.play_buf.get_win_phase()), String(self.play_buf.get_phase()))

        out = out * self.is_playing * min_env(self.play_buf.get_win_phase(), self.note_time, 0.01)

        return out

    fn get_msgs(mut self: Self):
        # Get messages from the world
        msg = self.world_ptr[0].get_msg("print_inputs")
        if msg:
            for i in range(self.world_ptr[0].num_in_chans):
                print("input[", i, "] =", self.world_ptr[0].sound_in[i])
        msg = self.world_ptr[0].get_msg("start_recording")
        if msg:
            self.write_pos = 0
            self.is_recording = 1.0
            self.is_playing = 0.0
            self.trig = 0.0
        msg = self.world_ptr[0].get_msg("set_input_chan")
        if msg:
            chan = Int64(msg.value()[0])
```

```
            if chan >= 0 and chan < self.world_ptr[0].num_in_chans:
                self.input_chan = chan
                print("Setting input channel to", chan)
        note_ons = self.world_ptr[0].get_midi("note_on",-1, -1)  # Get all note on messages
        if note_ons:
            self.note_ons = note_ons.value().copy()
        note_offs = self.world_ptr[0].get_midi("note_off",-1, -1)  # Get all note off messages
        if note_offs:
            self.note_offs = note_offs.value().copy()

# there can only be one graph in an MMMAudio instance
# a graph can have as many synths as you want
struct Record(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]

    var synth: Record_Synth

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.synth = Record_Synth(self.world_ptr)

    fn __repr__(self) -> String:
        return String("Record")

    fn next(mut self) -> SIMD[DType.float64, 2]:
        sample = self.synth.next()
        # print("sample:", sample)
        return sample
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.9 Midi_Sequencer

4.9.1 `trig_synth(wait)` `async`

A counter coroutine

Source code in `examples/Midi_Sequencer.py`

```
async def trig_synth(wait):
    """A counter coroutine"""
    count_to = np.random.choice([7, 11, 13, 17]).item()
    mult_seq = Pseq(list(range(1, count_to + 1)))
    fund_seq = Pxrand([36, 37, 43, 42])
    i = 0
    fund = librosa.midi_to_hz(fund_seq.next())
    while True:
        pitch = mult_seq.next() * fund
        mmm_audio.send_msg("t_trig", 1.0)
        mmm_audio.send_msg("trig_seq_freq", pitch)
        await asyncio.sleep(wait)
        i = (i + 1) % count_to
        if i == 0:
            fund = librosa.midi_to_hz(fund_seq.next())
            count_to = np.random.choice([7, 11, 13, 17]).item()
            mult_seq = Pseq(list(range(1, count_to + 1)))
```

4.9.2 Python Code

```
from mmm_src.MMMAudio import MMMAudio

# instantiate and load the graph
mmm_audio = MMMAudio(128, graph_name="Midi_Sequencer", package_name="examples")
mmm_audio.start_audio()

# this next chunk of code is all about using a midi keyboard to control the synth---------------

# the python host grabs the midi and sends the midi messages to the mojo audio engine

import mido
import time
import threading

# find your midi devices
mido.get_input_names()

# open your midi device - you may need to change the device name
in_port = mido.open_input('Oxygen Pro Mini USB MIDI')

def start_midi():
    while True:
        for msg in in_port.iter_pending():
            # print(msg)
            mmm_audio.send_midi(msg)
        time.sleep(0.01) # Small delay to prevent busy-waiting

midi_thread = threading.Thread(target=start_midi, daemon=True)
# once you start the midi_thread, it should register note_on, note_off, cc, etc from your device and send them to mmm
midi_thread.start()
midi_thread.stop()

# this chunk of code shows how to use the sequencer to trigger notes in the mmm_audio engine

# the scheduler can also sequence notes
from mmm_src.Patterns import * # some sc style patterns
import numpy as np
import asyncio
import librosa

scheduler = mmm_audio.scheduler

async def trig_synth(wait):
    """A counter coroutine"""
    count_to = np.random.choice([7, 11, 13, 17]).item()
    mult_seq = Pseq(list(range(1, count_to + 1)))
    fund_seq = Pxrand([36, 37, 43, 42])
    i = 0
    fund = librosa.midi_to_hz(fund_seq.next())
```

```
    while True:
        pitch = mult_seq.next() * fund
        mmm_audio.send_msg("t_trig", 1.0)
        mmm_audio.send_msg("trig_seq_freq", pitch)
        await asyncio.sleep(wait)
        i = (i + 1) % count_to
        if i == 0:
            fund = librosa.midi_to_hz(fund_seq.next())
            count_to = np.random.choice([7, 11, 13, 17]).item()
            mult_seq = Pseq(list(range(1, count_to + 1)))

scheduler.sched(trig_synth(0.1))

scheduler.stop_routs()

mmm_audio.stop_audio()
mmm_audio.start_audio()
```

4.9.3 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld

from mmm_utils.functions import *
from mmm_dsp.Osc import *
from mmm_dsp.Filters import *
from mmm_dsp.Env import Env

from mmm_src.MMMTraits import *


struct TrigSynthVoice(Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]  # Pointer to the MMMWorld instance

    var env: Env

    var mod: Osc
    var car: Osc
    var lag: Lag

    var trig: Float64
    var freq: Float64

    var vol: Float64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr

        self.mod = Osc(self.world_ptr)
        self.car = Osc(self.world_ptr)
        self.lag = Lag(self.world_ptr)

        self.env = Env(self.world_ptr)

        self.trig = 0.0
        self.freq = 100.0
        self.vol = 1.0

    fn next(mut self) -> Float64:
        if not self.env.is_active and self.trig <= 0.0:
            return 0.0  # Return 0 if the envelope is not active and no trigger
        else:
            var mod_value = self.mod.next(self.freq*1.5)  # Get the next value from the modulator
            var env = self.env.next([0.0, 1.0, 0.75, 0.75, 0.0], [0.01, 0.1, 0.2, 0.5], [1.0], 0, self.trig)
            var mod_mult = linlin(env, 0.0, 1.0, 0.0, 0.25) # self.lag.next(linlin(self.mouse_x, 0.0, 1.0, 0.0, 8.0), 0.05)
            var car_value = self.car.next(self.freq, mod_value * mod_mult, osc_type=2, os_index=1)  # Get the next value from the carrier
            car_value = car_value * 0.1 * env * self.vol

            return car_value

struct TrigSynth(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]  # Pointer to the MMMWorld instance

    var voices: List[TrigSynthVoice]
    var current_voice: Int64
    var trig: Float64
    var freq: Float64
    var num_voices: Int64
    var note_ons: List[List[Int64]]
    var ccs: List[List[Int64]]

    var svf: SVF
    var filt_lag: Lag
    var filt_freq: Float64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld], num_voices: Int64 = 8):
        self.world_ptr = world_ptr
        self.trig = 0.0
        self.freq = 100.0
        self.num_voices = num_voices
        self.current_voice = 0
```

```
        self.note_ons = List[List[Int64]]()
        self.ccs = List[List[Int64]]()

        self.voices = List[TrigSynthVoice]()
        for _ in range(self.num_voices):
            self.voices.append(TrigSynthVoice(self.world_ptr))

        self.svf = SVF(self.world_ptr)
        self.filt_lag = Lag(self.world_ptr)
        self.filt_freq = 1000.0

    fn __repr__(self) -> String:
        return String("OscSynth")

    fn next(mut self) -> SIMD[DType.float64, 2]:
        self.get_msgs()

        var out = 0.0

        for note_on in self.note_ons:
            print(note_on[0], note_on[1], note_on[2], end = "\n")
            self.current_voice = (self.current_voice + 1) % self.num_voices
            self.voices[self.current_voice].vol = Float64(note_on[2]) / 127.0
            self.voices[self.current_voice].trig = 1.0
            self.voices[self.current_voice].freq = midicps(note_on[1])
        self.note_ons.clear()

        # looking for midi cc on cc 34
        # this will control the frequency of the filter
        for cc in self.ccs:
            if cc[1] == 34:  # Assuming CC 34 is for filter frequency
                self.filt_freq = linlin(Float64(cc[2]), 0.0, 127.0, 20.0, 1000.0)  # Map CC value to frequency range

        if self.trig > 0.0:
            self.current_voice = (self.current_voice + 1) % self.num_voices
            self.voices[self.current_voice].trig = self.trig
            self.voices[self.current_voice].freq = self.freq

        # get the output of all the synths and reset the of the current voice (after getting audio)
        for i in range(len(self.voices)):
            out += self.voices[i].next()
            self.trig = 0.0
            self.voices[i].trig = 0.0  # Reset the trigger for the next iteration

        out = self.svf.lpf(out, self.filt_lag.next(self.filt_freq, 0.1), 2.0) * 0.6

        return out

    fn get_msgs(mut self: Self):
        # calls to get_msg and get_midi return an Optional type
        # so you must get the value, then test the value to see if it exists, before using the value
        # get_msg returns a single list of values while get_midi returns a list of lists of values

        trig = self.world_ptr[0].get_msg("t_trig") # trig will be an Optional
        if trig: # if it trig is None, we do nothing
            self.trig = trig.value()[0]
        freq = self.world_ptr[0].get_msg("trig_seq_freq")
        if freq:
            self.freq = freq.value()[0]
        note_ons = self.world_ptr[0].get_midi("note_on",-1, -1)  # Get all note on messages
        if note_ons:
            self.note_ons = note_ons.value().copy()

        ccs = self.world_ptr[0].get_midi("control_change",-1, -1)
        if ccs:
            self.ccs = ccs.value().copy()

struct Midi_Sequencer(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]

    var output: List[Float64]  # Output buffer for audio samples

    var trig_synth: TrigSynth  # Instance of the Oscillator

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.output = List[Float64](0.0, 0.0)  # Initialize output list

        self.trig_synth = TrigSynth(world_ptr)  # Initialize the TrigSynth with the world instance

    fn __repr__(self) -> String:
        return String("Midi_Sequencer")

    fn next(mut self: Midi_Sequencer) -> SIMD[DType.float64, 2]:
        return self.trig_synth.next()  # Return the combined output sample
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.10 Torch_MLP

this examples uses a Torch MLP model to control a 16 parameter synth to play the synth, just hang out in the top 4 lines of code and play with the mouse

you can also train the synth by creating any number of input/output pairs and making a new training

4.10.1 Python Code

```
from mmm_src.MMMAudio import MMMAudio
from random import random

mmm_audio = MMMAudio(128, graph_name="Torch_MLP", package_name="examples")

mmm_audio.start_audio() # start the audio thread - or restart it where it left off

mmm_audio.stop_audio()  # stop/pause the mojo thread


# below is the code to make a new training --------------------------------


# if you make a new training below, you can load it into the synth
mmm_audio.send_text_msg("load_mlp_training", "examples/nn_trainings/model_traced.pt")


# toggle inference off so you can set the synth values directly
mmm_audio.send_msg("toggle_inference", 1.0)

out_size = 16

def make_setting():
    setting = []
    for i in range(out_size):
        setting.append(random())
        mmm_audio.send_msg("model_output" + str(i), setting[i])

    return setting

outputs = make_setting()

X_train_list = []
y_train_list = []

for i in range(len(y_train_list)):
    print(f"Element {i}: {X_train_list[i]}")
    print(f"Element {i}: {y_train_list[i]}")

# when you like a setting add an input and output pair
# this is assuming you are training on 4 pairs of data points
X_train_list.append([0,0])
y_train_list.append(outputs)

X_train_list.append([0,1])
y_train_list.append(outputs)

X_train_list.append([1,1])
y_train_list.append(outputs)

X_train_list.append([1,0])
y_train_list.append(outputs)

learn_rate = 0.001
epochs = 5000

layers = [ [ 64, "relu" ], [ 64, "relu" ], [ out_size, "sigmoid" ] ]

from mmm_utils.mlp_trainer import train_nn

train_nn(X_train_list, y_train_list, layers, learn_rate, epochs, "examples/nn_trainings/model_traced.pt")
```

4.10.2 Mojo Code

```
from mmm_src.MMMWorld import MMMWorld
from mmm_utils.functions import *
from mmm_src.MMMTraits import *

# THE SYNTH - is imported from TorchSynth.mojo in this directory
from .TorchSynth import TorchSynth
```

```
# THE GRAPH

struct Torch_MLP(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var torch_synth: TorchSynth  # Instance of the TorchSynth

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr

        self.torch_synth = TorchSynth(world_ptr)  # Initialize the TorchSynth with the world instance

    fn __repr__(self) -> String:
        return String("Torch_MLP")

    fn next(mut self: Torch_MLP) -> SIMD[DType.float64, 2]:

        return self.torch_synth.next()
```

For more information about the examples, such as how the Python and Mojo files interact with each other, see the Examples Overview

4.11 OleDusty

4.11.1 Python Code

```python
from mmm_src.MMMAudio import MMMAudio

# instantiate and load the graph
mmm_audio = MMMAudio(128, graph_name="OleDusty", package_name="examples")
mmm_audio.start_audio()
```

4.11.2 Mojo Code

```mojo
from mmm_src.MMMWorld import MMMWorld
from mmm_dsp.Osc import Dust
from mmm_utils.functions import *
from mmm_dsp.Filters import *

# THE SYNTH

struct Dusty(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var dust: Dust[2]

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.dust = Dust[2](world_ptr)

    fn __repr__(self) -> String:
        return String("OleDusty")

    fn next(mut self, freq: Float64) -> SIMD[DType.float64, 2]:

        out = self.dust.next(freq)

        # uncomment below for use the phase of the Dust oscillator instead of the impulse
        # out = self.dust.get_phase()

        return out

# THE GRAPH

struct OleDusty(Representable, Movable, Copyable):
    var world_ptr: UnsafePointer[MMMWorld]
    var dusty: Dusty
    var reson: Reson[2]
    var freq: Float64

    fn __init__(out self, world_ptr: UnsafePointer[MMMWorld]):
        self.world_ptr = world_ptr
        self.dusty = Dusty(world_ptr)
        self.reson = Reson[2](world_ptr)
        self.freq = 20.0

    fn __repr__(self) -> String:
        return String("OleDusty")

    fn next(mut self) -> SIMD[DType.float64, 2]:

        freq = linexp(self.world_ptr[0].mouse_y, 0.0, 1.0, 100.0, 2000.0)

        out = self.dusty.next(linlin(self.world_ptr[0].mouse_x, 0.0, 1.0, 5.0, 200.0))

        # there is really no difference between ugens, synths, graphs
        # thus there is no reason you can't process the output of a synth directly in the graph
        # the reson filter uses SIMD to run 2 filters in parallel, each processing a channel of the dusty synth
        out = self.reson.hpf(out, freq, 10.0, 1.0)  # apply a bandpass filter to the output of the Dusty synth

        return out
```

5. Contributing

5.1 Contributing to MMMAudio

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

5.1.1 Development Setup

Prerequisites

See the README.md file in the repo's root directory for how to get MMMAudio up and running.

5.1.2 Contributing Guidelines

Code Style

PYTHON CODE

- Use type hints for all function signatures
- Use Google-style docstrings

MOJO CODE

- Follow Mojo style conventions
- Use SIMD types for performance-critical code

Documentation

- All public APIs must be documented
- Include practical examples for each function
- Update documentation when changing functionality
- Consult the documentation examples/style-guides in `doc_generation/examples`

Code of Conduct

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

5.2 Documentation for MMMAudio Repo

This repo uses mkdocs to render code documentation files from Google-style docstrings in Python and Mojo files.

5.2.1 Dependencies

See root/requirements-docs.txt.

5.2.2 What the `doc_generation` directory contains:

- **examples/**: An example Python and Mojo file for how to put the in-source-file documentation into the respective language files. These shouldn't be modified, just use as a reference / style guide.
- **static_docs/**: This directory contains any documentation files that are *not* generated from source code or examples. This includes things like "Getting Started" and "Contributing". To edit this content, edit these Markdown files directly. This directory also maintains the directory structure that will be *copied* into the docs_md/ directory. The docs_md/ directory is the actual directory the mkdocs will look to to generate HTML from Markdown files, which is to say that the directory structure of the HTML files and website comes from docs_md/ which is copied form static_docs/, so maintaining this directory structure is important. You'll notice there are `index.md` files, these are the "homepages" for each directory (including the site's top level directory).
- **templates/**: Contains jinja2 templates that are used for rendering Markdown files.
- *mojo_doc_template_jinja.md*: Used to inject the contents of a `json` generated by `mojo doc` (which becomes a `dict` in `generate_docs.py`) into Markdown format.
- *example_python_and_mojo_jinja.md*: Template for making a page for each example in the root/examples directory. In order to render properly, this template (and the corresponding Python code in `generate_docs.py` expect each example to consist of two files: (1) a Python file and (2) a Mojo file with **the same base name**). Examples should be constructed using only two files using this convention.
- *struct.md*: A partial template for rendering a Mojo `struct`. This partial template is "called" in `mojo_doc_template_jinja.md`.
- **init.py**: Makes this directory a Python package so that the `main()` function of the `generate_docs.py` file can be called as a "hook"
- **generate_docs.py**: Python script that creates directory structure and generates Markdown files from jinja2 templates into the `docs_md` directory in preparation for mkdocs to use `docs_md` directory to create HTML (which ends up in the `root/docs` directory for GitHub Pages to use).

5.2.3 Building Documentation / How It Works

Presented here in "chronological" order

1. Make sure to install dependencies:

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

1. In the root directory, run `mkdocs build`
2. The `mkdocs.yml` file indicates that the `on_pre_build` hook should run, which is the `main()` function of `generate_docs.py`. This script:
3. Clears out the contents of the `docs_md` directory so there are no stale documents lingering there that would unintentionally be rendered by mkdocs
4. Copies the contents of `static_docs` into `docs_md` to establish the directory structure and provide the Markdown files that are not generated from source or examples.
5. Finds all the Mojo files in directories that contain source files (specified in `generate_docs.py` with the variable `HARDCODED_SOURCE_DIRS`) and for each file: (a) uses `mojo doc` to get a `json` string from standard out, (b) turn that string in to `dict`, (c) removes information from that `dict` that isn't worth rendering in the documentation, such as the methods `__init__` and `__repr__` as well as the argument `self`, (d) uses the remaining contents of the `dict` as context for rendering a Markdown file to document what is in the file. The Markdown file has the same basename as the Mojo file. **Because the Mojo file basename corresponds to the way it appears in the documentation, each Mojo struct should live in its own file. This will make the documentation clearer to navigate on the documentation site.**
6. Looks in the `root/examples` directory and finds all the Python files. **It assumes there will be a Mojo file of the same name. The script also assumes that each example consists of just the two files and that any other code that is needed can be imported from the MMMAudio core.** This correspondence simplifies file management for rendering the examples into the documentation and makes the process of editing and creating examples clearer. The two files are both pasted into a Markdown file (using `example_python_and_mojo_jinja.md`) which is saved to the `docs_md/examples` directory.
7. Once `generate_docs.py` is complete, `mkdocs` then build the site, putting all the HTML in the `docs` directory.

5.2.4 Serve Locally

To preview the documentation locally:

```
mkdocs serve
```

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

5.2.5 Build PDF

A PDF version of the docs is automatically built when `mkdocs build` is run. To turn this off, remove the `with-pdf` plug-ing from the `mkdocs.yml` file.