torchsynth

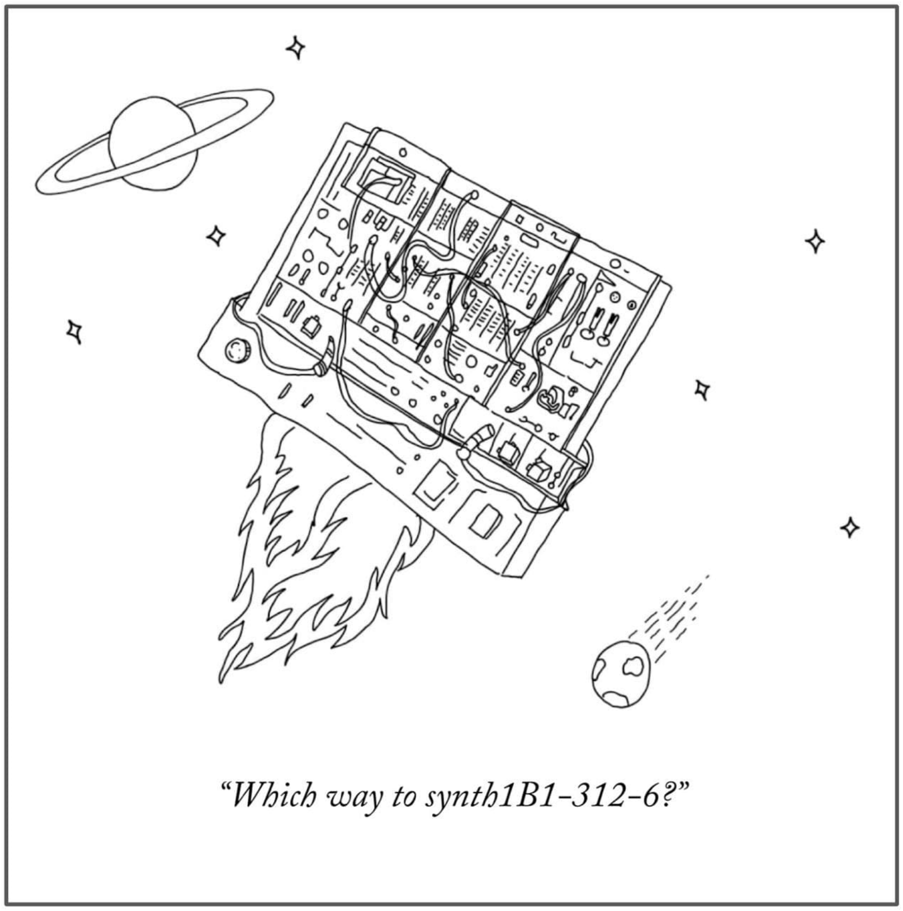
Release 1.0.1

Jordie Shier, Joseph Turian, Max Henry

Sep 12, 2021

GETTING STARTED

The fastest synth in the universe.



torchsynth is based upon traditional modular synthesis written in pytorch. It is GPU-optional and differentiable.

Most synthesizers are fast in terms of latency. torchsynth is fast in terms of throughput. It synthesizes audio 16200x faster than realtime (714MHz) on a single GPU. This is of particular interest to audio ML researchers seeking large training corpora.

Additionally, all synthesized audio is returned with the underlying latent parameters used for generating the corresponding audio. This is useful for multi-modal training regimes.

If you'd like to hear torchsynth, check out synth1K1, a dataset of 1024 4-second sounds rendered from the `Voice` synthesizer, or listen to the following SoundCloud embed:

Here is another set of sounds created with the `Voice` *Drum Nebula*. In torchsynth a *nebula* is a set of hyperparameters that defines how synthesizer parameters are sampled. The hyperparameters of the Drum Nebula were hand-tuned to increase the likelihood of the `Voice` producing percussive samples.

CHAPTER

ONE

INSTALLATION

```
pip3 install torchsynth
```

Note that torchsynth requires PyTorch version 1.8 or greater.

CHAPTER

TWO

QUICKSTART

Contents

2.1 Which way to synth1B1-312-6?

In this simple example, we use Voice to generate the 312th batch of synth1B1. This batch comprises audio, parameters, and whether the instances are training examples. We then select sample 6 from the audio batch, and save it to a WAV file.

You will need to `pip install torchaudio` in order to save the WAV file. Alternately, you could modify the code slightly and use SoundFile.

```
import torch
import torchaudio
from torchsynth.synth import Voice

voice = Voice()
# Run on the GPU if it's available
if torch.cuda.is_available():
    voice = voice.to("cuda")

# Generate batch 312
# All audio batches are [128, 176400], i.e. 128 4-second sounds at 44100Hz
# Each sound is a monophonic 1D tensor.
# Param batches are [128, 72], which are the 72 latent Voice
# parameters that generated each sound.
# The training tensor is a [128] bool, indicating whether
# instances are designated as train or test, for reproducibility.
synth1B1_312_audio, synth1B1_312_params, synth1B1_312_is_train = voice(312)

# Select synth1B1-312-6
synth1B1_312_6 = synth1B1_312_audio[6]

# We add one channel at the beginning, for torchaudio
torchaudio.save("synth1B1-312-6.wav", synth1B1_312_6.unsqueeze(0).cpu(), voice.sample_
↪rate)
```

CHAPTER THREE

DETAILED WALKTHROUGH

`examples/examples.ipynb` (and corresponding `examples/examples.py`) contains a detailed walkthrough of the code.

You can also try it in Colab.

CHAPTER

FOUR

BATCH PROCESSING AND PERFORMANCE

Contents

4.1 Batch Processing

To take advantage of the parallel processing power of a GPU, all modules render audio in batches. Larger batches enable higher throughput on GPUs. The default batch size is 128, which requires ≈2.3GB of GPU memory, and is 16200x faster than realtime on a V100. (GPU memory consumption is approximately $\approxeq 1216 + 8.19 \cdot \text{batch\_size}$ MB, including the torchsynth model.)

4.2 ADSR Batches

An example of a batch of 4 of randomly generated ADSR envelopes is shown below:

CHAPTER

FIVE

MULTI-GPU SUPPORT

Multiple GPUs are supported. More documentation forthcoming.

CHAPTER

SIX

MODULAR PRINCIPLES

Contents

6.1 Synth Modules

The design of torchsynth is inspired by hardware modular synthesizers which contain individual units. Each module has a specific function and parameters, and they can be connected together in various configurations to construct a synthesizer. There are three types of modules in torchsynth: audio modules, control modules, and parameter modules. Audio modules operate at audio sampling rate (default 44.1kHz) and output audio `Signal`. Examples include voltage-controlled oscillators (`VCO`) and voltage-controlled amplifiers (`VCA`s). Control modules output control signals that are used modulate the parameters of another module. For speed, these modules operate at a reduced control rate (default 441Hz). Examples of control modules include `ADSR` envelope generators and low frequency oscillators (`LFO`s). Finally, parameter modules simply output parameters. Examples of these include a `MonophonicKeyboard` module that has no input, and outputs the note midi f0 value and duration.

6.2 Synth Architectures

The default configuration in torchsynth is the `Voice`, which is the architecture used in synth1B1. The `Voice` architecture comprises the following modules: a `MonophonicKeyboard`, two `LFO`, six `ADSR` envelopes (each `LFO` module includes two dedicated `ADSR`: one for rate modulation and another for amplitude modulation), one `SineVCO`, one `SquareSawVCO`, one `Noise` generator, `VCA`, a `ModulationMixer` and an `AudioMixer`. Modulation signals generated from control modules (`ADSR` and `LFO`) are upsampled to the audio sample rate before being passed to audio rate modules.

The figure below shows the configuration and routing of the modules composing `Voice`.

CHAPTER

SEVEN

EXAMPLE OF HOW TO BUILD NEW SYNTHS

In this example we'll create a new synthesizer using modules (*SynthModule*). Synths in torchsynth are created using the approach modular synthesis that involves connecting individual modules. We'll create a simple single oscillator synth with an attack-decay-sustain-release (*ADSR*) envelope controlling the amplitude. More complicated architectures can be created using the same ideas.

You can also view this example in Colab.

7.1 Creating the SimpleSynth class

All synths in torchsynth derive from *AbstractSynth*, which provides helpful functionality for managing children *SynthModule*s and their *ModuleParameter*s.

There are two steps involved in creating a class that derives from *AbstractSynth*:

1. The `__init__` method instantiates the *SynthModule*s that will be used.
2. The *output()* method defines how individual *SynthModule*s are connected: Which modules' output is the input to other modules, and the final output.
3. *forward* wraps `output`, ensuring reproducibility if desired.

7.1.1 Defining the modules

Here we create our `SimpleSynth` class that derives from *AbstractSynth*. Override the `__init__` method and include an optional parameter for *SynthConfig*. *SynthConfig* holds the global configuration information for the synth and its modules, including the batch size, sample rate, buffer rate, etc.

To register modules for use within `SimpleSynth`, we pass them in as a list to the class method *add_synth_modules()*. This list contains tuples with the name that we want to have for the module in the synth as well as the `SynthModule`. Each module passed in this list will be instantiated using the same *SynthConfig* object and added as a class attribute with the name defined by the first item in the tuple.

```python
from typing import Optional
import torch
from torchsynth.synth import AbstractSynth
from torchsynth.config import SynthConfig
from torchsynth.module import (
    ADSR,
    ControlRateUpsample,
    MonophonicKeyboard,
    SquareSawVCO,
    VCA,
```

(continues on next page)

(continued from previous page)

```
)

class SimpleSynth(AbstractSynth):

    def __init__(self, synthconfig: Optional[SynthConfig] = None):

        # Call the constructor in the parent AbstractSynth class
        super().__init__(synthconfig=synthconfig)

        # Add all the modules that we'll use for this synth
        self.add_synth_modules(
            [
                ("keyboard", MonophonicKeyboard),
                ("adsr", ADSR),
                ("upsample", ControlRateUpsample),
                ("vco", SquareSawVCO),
                ("vca", VCA),
            ]
        )
```

7.1.2 Connecting Modules

Now that we have registered the modules that we are going to use. We define how they all are connected together in the overridden `output()` method.

```
    def output(self) -> torch.Tensor:
         # Keyboard is parameter module, it returns parameter
        # values for the midi_f0 note value and the duration
        # that note is held for.
        midi_f0, note_on_duration = self.keyboard()

        # The amplitude envelope is generated based on note duration
        envelope = self.adsr(note_on_duration)

        # The envelope that we get from ADSR is at the control rate,
        # which is by default 100x less than the sample rate. This
        # reduced control rate is used for performance reasons.
        # We need to upsample the envelope prior to use with the VCO output.
        envelope = self.upsample(envelope)

        # Generate SquareSaw output at frequency for the midi note
        out = self.vco(midi_f0)

        # Apply the amplitude envelope to the oscillator output
        out = self.vca(out, envelope)

        return out
```

7.2 Playing our SimpleSynth

That's out simple synth! Let's test it out now.

If we instantiate `SimpleSynth` without passing in a *SynthConfig* object then it will create one with the default options. We don't need to render a full batch size for this example, so let's use the smallest batch size that will support reproducible output. All the parameters in a synth are randomly assigned values, with reproducible mode on, we pass a batch_id value into our synth when calling it. The same sounds will always be returned for the same batch_id.

```
from torchsynth.config import BASE_REPRODUCIBLE_BATCH_SIZE

# Create SynthConfig with smallest reproducible batch size.
# Reproducible mode is on by default.
synthconfig = SynthConfig(batch_size=BASE_REPRODUCIBLE_BATCH_SIZE)
synth = SimpleSynth(synthconfig)

# If you have access to a GPU.
if torch.cuda.is_available():
    synth.to("cuda")
```

Now, let's make some sounds! We just call synth with a batch_id.

```
audio = synth(0)
```

Here are the results of the first 32 sounds concatenated together. Each sound is four seconds long and was generated by randomly sampling the parameters of SimpleSynth.

CHAPTER

EIGHT

REPRODUCIBLITY

Contents

8.1 Overview

We use deterministic random number generation to ensure replicability, even of noise oscillators. Nonetheless, there are small numeric differences between the CPU and GPU. The mean-average-error between audio samples generated on CPU and GPU are < 1e-2.

8.2 Defaults

Reproducibility is currently guaranteed when `batch_size` is multiple of 32 and you use the default *`SynthConfig`* settings: `sample_rate`=44100, `control_rate`=441.

8.3 Train vs Test

If a train/test split is desired, 10% of the samples are marked as test. Because researchers with larger GPUs seek higher-throughput with batchsize 1024, $9 \cdot 1024$ samples are designated as train, the next 1024 samples as test, etc.

All *`AbstractSynth forward()`* methods return three batched tensors: audio, latent parameters, and an `is_train` boolean vector.

CHAPTER

NINE

SYNTH1B1

Contents

9.1 Overview

synth1B1 is a corpus consisting of one million hours of audio: one billion 4-second synthesized sounds. The corpus is multi-modal: Each sound includes its corresponding synthesis parameters.

synth1B1 is generated *on the fly* by torchsynth 1.x, using the Voice synth with its default settings.

9.2 Experimental Control

Researchers can denote subsamples of this corpus as synth1M1, synth10M1, *etc.*, which would refer to the first 1 million and 10 million samples of Synth1B1 respectively.

Besides having a `batch_size` that is a multiple of 32, if you change any of the defaults in *`SynthConfig`*, please call that in your work, and use a variant of the name synth1B1.

One tenth of the examples are designated as the test set. See *Reproducibility > Train vs. Test* for more information.

The nomenclature "synth1B1-312-6" denotes batch 312 (assuming batch size of 128) and sound 6 within that batch.

9.3 Semantic Versioning

We use a slightly different convention than traditional Semantic Versioning.

Given a version number MAJOR.MINOR.PATCH, we increment the:

- MAJOR version when the default output of Voice changes.
- MINOR version when we make incompatible API changes, but the default output of Voice remains reproducible.
- PATCH version when we make backwards compatible bug fixes and improvements.

For example, any torchsynth 1.x release can generate synth1B1.

When torchsynth 2.x is released, it will generate synth1B2. Any pre-release (*e.g.* 2.0.0-pre1) is *not* guaranteed to generate synth1B2 until 2.0.0 is released.

CHAPTER

TEN

DEVELOPER GUIDELINES

Contents

10.1 Initial developer setup

```
git clone https://github.com/torchsynth/torchsynth
cd torchsynth
pip3 install -e ".[dev]"
```

Make sure you have pre-commit hooks installed:

```
pre-commit install
```

This helps us avoid checking dirty jupyter notebook cells into the repo.

10.2 Examples

In Python or Jupyter Notebook (if you want to see pretty plots), run `examples/examples.py`.

10.2.1 Python 3.9 on OSX

Unfortunately, Python 3.9 on OSX Big Sur won't work, because librosa repends upon numba which isn't packaged for 3.9 yet. In which case you'll have to create a Python 3.7 conda environment. (You might also need to downgrade LLVM to 10 or 9.):

```
conda install -c conda-forge ipython librosa matplotlib numpy matplotlib scipy␣
↪jupytext
conda install -c anaconda ipykernel
python -m ipykernel install --user --name=envname
```

and change the kernel to `envname`.

10.3 Tests

Unit testing is performed using `pytest`.

To run tests, run `pytest` from the project root:

```
TORCHSYNTH_DEBUG=True pytest
```

To run tests with a coverage report:

```
pytest --cov=./torchsynth
```

Examples also serves as an integration test:

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

CHAPTER

ELEVEN

TORCHSYNTH.CONFIG

Global configuration for *AbstractSynth* and its component *SynthModule*.

torchsynth.config.**BASE_REPRODUCIBLE_BATCH_SIZE = 32**
Smallest batch size divisor that is supported for any reproducible output This is because `Noise`: creates deterministic noise batches in advance, for speed.

torchsynth.config.**DEFAULT_BATCH_SIZE = 128**
This batch size is a nice trade-off between speed and memory consumption. On a typical GPU this consumes ~2.3GB of memory for the default Voice. Learn more about batch processing.

torchsynth.config.**N_BATCHSIZE_FOR_TRAIN_TEST_REPRODUCIBILITY = 1024**
If a train/test split is desired, 10% of the samples are marked as test. Because researchers with larger GPUs seek higher-throughput with batchsize 1024, $9 cdot 1024$ samples are designated as train, the next 1024 samples as test, etc.

class torchsynth.config.**SynthConfig**(*batch_size=128*, *sample_rate=44100*, *buffer_size_seconds=4.0*, *control_rate=441*, *reproducible=True*, *no_grad=True*, *debug=False*, *eps=1e-06*)

Bases: `object`

Any *SynthModule* and *AbstractSynth* might use these global configuration values. Every *SynthModule* in the same *AbstractSynth* should have the save SynthConfig.

Parameters

- **batch_size** (`int`) – Scalar that indicates how many parameter settings there are, i.e. how many different sounds to generate.
- **sample_rate** (`Optional[int]`) – Scalar sample rate for audio generation.
- **buffer_size** – Duration of the output in seconds.
- **control_rate** (`Optional[int]`) – Scalar sample rate for control signal generation. reproducible: Reproducible results, with a small performance impact.
- **no_grad** (`bool`) – Disables gradient computations.
- **debug** (`bool`) – Run slow assertion tests. (Default: False, unless environment variable TORCHSYNTH_DEBUG exists.)
- **eps** (`float`) – Epsilon to avoid log underrun and divide by zero.

to(*device*)
For speed, we've noticed that it is only helpful to have sample and control rates on device, and as a float.

torchsynth.config.**check_for_reproducibility**()
This method is called automatically if your *SynthConfig* specifies `reproducibility=True`.

Reproducible results are important to torchsynth and synth1B1, so we are testing to make sure that the expected random results are produced by torch.rand when seeded. This raises an error indicating if reproducibility is not guaranteed.

Running torch.rand on CPU and GPU give different results, so all seeded randomization where reproducibility is important occurs on the CPU and then is transferred over to the GPU, if one is being used. See https://discuss.pytorch.org/t/deterministic-prng-across-cpu-cuda/116275

torchcsprng allowed for determinism between the CPU and GPU, however profiling indicated that torch.rand on CPU was more efficient. See https://github.com/pytorch/csprng/issues/126

CHAPTER

TWELVE

TORCHSYNTH.MODULE

12.1 SynthModule

class `torchsynth.module.`**`SynthModule`**(*synthconfig*, *device=None*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

A base class for synthesis modules. A *`SynthModule`* optionally takes input from other *`SynthModule`* instances. The *`SynthModule`* uses its (optional) input and its set of *`ModuleParameter`* to generate output. All *`ModuleParameter`* of the *`SynthModule`* are assumed to be `batch_size`-length 1-D tensors.

All *`SynthModule`* objects should be atomic, i.e., they should not contain other *`SynthModule`* objects. This design choice is in the spirit of modular synthesis.

Parameters

- **synthconfig** (*`SynthConfig`*) – An object containing synthesis settings that are shared across all modules, typically specified by *`Voice`*, or some other, possibly custom *`AbstractSynth`* subclass.
- **device** (`Optional[device]`) – An object representing the device on which the *torch* tensors are to be allocated (as per PyTorch, broadly).

add_parameters(*parameters*)

Adds parameters to the *`SynthModule`* parameter dictionary. Used by the class constructor.

Parameters **parameters** (`List[`*`ModuleParameter`*`]`) – List of parameters to register with this module.

property batch_size

Size of the batch to be generated.

Return type `Tensor`

property buffer_size

Size of the module output in samples.

Return type `Tensor`

property eps

A very small value used to avoid computational errors.

Return type `float`

forward(**args*, ***kwargs*)

Wrapper for output that ensures a *`buffer_size`* length output.

Return type *`Signal`*

get_parameter(*parameter_id*)
Retrieves a single *ModuleParameter*, as specified by its parameter Id.

Parameters **parameter_id** (str) – Id of the parameter to retrieve.

Return type *ModuleParameter*

get_parameter_0to1(*parameter_id*)
Retrieves a specified parameter value in the normalized range [0,1].

Parameters **parameter_id** (str) – Id of the parameter to retrieve.

Return type Tensor

property nyquist
Convenience property for the highest frequency that can be represented at *sample_rate* (as per Shannon-Nyquist).

output(**args*, ***kwargs*)
Performs the main action of *SynthModule*. Each child class should override this method.

Return type *Signal*

p(*parameter_id*)
Convenience method for retrieving a parameter value. Returns the value in parameter-specific, non-normalized range.

Parameters **parameter_id** (str) – Id of the parameter to retrieve.

Return type Tensor

property sample_rate
Sample rate frequency in Hz.

Return type Tensor

seconds_to_samples(*seconds*)
Convenience function to calculate the number of samples corresponding to given a time value and *sample_rate*. Returns a possibly fractional value.

Parameters **seconds** (Tensor) – Time value in seconds.

Return type Tensor

set_parameter(*parameter_id*, *value*)
Updates a parameter value in a parameter-specific non-normalized range.

Parameters

- **parameter_id** (str) – Id of the parameter to update.
- **value** (Tensor) – Value to assign to the parameter.

set_parameter_0to1(*parameter_id*, *value*)
Update a parameter value in a normalized range [0,1].

Parameters

- **parameter_id** (str) – Id of the parameter to update.
- **value** (Tensor) – Value to assign to the parameter.

to(*device=None*, ***kwargs*)
This function overrides the `to()` call in `torch.nn.Module`. It ensures that the related values *ModuleParameterRange* and *ModuleParameter*, as well as `synthconfig` are also transferred to the correct device.

Parameters **`device`** (`Optional[device]`) – device to send this module to

`to_buffer_size`(*signal*)

Fixes the length of a signal to the default buffer size of this module, as specified by *`buffer_size`*. Longer signals are truncated to length; shorter signals are zero-padded.

Parameters **`signal`** (*`Signal`*) – A signal to pad or truncate.

Return type *`Signal`*

12.2 Audio Rate Modules

These modules operate at full audio sampling rate.

12.2.1 AudioMixer

`class` `torchsynth.module.`**`AudioMixer`**(*synthconfig*, *n_input*, *curves=None*, *names=None*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

Sums together N audio signals and applies range-normalization if the resulting signal is outside of [-1, 1].

`output`(**signals*)

Returns a mixed signal from an array of input signals.

Return type *`Signal`*

12.2.2 ControlRateUpsample

`class` `torchsynth.module.`**`ControlRateUpsample`**(*synthconfig*, *device=None*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

Upsample control signals to the global sampling rate

Uses linear interpolation to resample an input control signal to the audio buffer size set in synthconfig.

`output`(*signal*)

Performs the main action of *`SynthModule`*. Each child class should override this method.

Return type *`Signal`*

12.2.3 Noise

`class` `torchsynth.module.`**`Noise`**(*synthconfig*, *seed*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

Generates white noise that is the same length as the buffer.

For performance noise is pre-computed. In order to maintain reproducibility noise must be computed on the CPU and then transferred to the GPU, if a GPU is being used. We pre-compute *`BASE_REPRODUCIBLE_BATCH_SIZE`* samples of noise and then repeat those for larger batch sizes.

To keep things fast we only support multiples of *`BASE_REPRODUCIBLE_BATCH_SIZE`* when reproducibility mode is enabled. For example, if you batch size is 4 times *`BASE_REPRODUCIBLE_BATCH_SIZE`*, then you get the same noise signals repeated 4 times.

Note: If you have multiple *Noise* modules in the same `AbstractSynth`, make sure you instantiate each *Noise* with a unique seed.

Parameters

- **`synthconfig`** (`SynthConfig`) – See `SynthModule`
- **`seed`** (`int`) – random number generator seed for white noise

`output`()

Performs the main action of `SynthModule`. Each child class should override this method.

Return type `Signal`

12.2.4 SineVCO

class torchsynth.module.**SineVCO**(*synthconfig*, *device=None*, ***kwargs*)

Bases: `torchsynth.module.VCO`

Simple VCO that generates a pitched sinusoid.

`oscillator`(*argument*, *midi_f0*)

A cosine oscillator. . . . Good ol' cosine.

Parameters

- **`argument`** (`Signal`) – The phase of the oscillator at each time sample.
- **`midi_f0`** (`Tensor`) – Fundamental frequency in midi (ignored in this VCO).

Return type `Signal`

12.2.5 SquareSawVCO

class torchsynth.module.**SquareSawVCO**(*synthconfig*, *device=None*, ***kwargs*)

Bases: `torchsynth.module.VCO`

An oscillator that can take on either a square or a sawtooth waveshape, and can sweep continuously between them, as determined by the `shape` parameter. A shape value of 0 makes a square wave; a shape of 1 makes a saw wave.

With apologies to Lazzarini and Timoney (2010). "New perspectives on distortion synthesis for virtual analog oscillators." Computer Music Journal 34, no. 1: 28-40.

Module Parameters

`tuning`

tuning adjustment for VCO in midi.

(**Min**: -24.0, **Max**: 24.0, **Curve**: 1, **Symmetric**: False)

`mod_depth`

depth of the pitch modulation in semitones.

(**Min**: -96.0, **Max**: 96.0, **Curve**: 0.2, **Symmetric**: True)

`initial_phase`

Initial phase for this oscillator.

(**Min**: -3.1415927410125732, **Max**: 3.1415927410125732, **Curve**: 1, **Symmetric**: False)

shape
Waveshape - square to saw [0,1].

(**Min**: 0.0, **Max**: 1.0, **Curve**: 1, **Symmetric**: False)

oscillator(*argument*, *midi_f0*)
Generates output square/saw audio given a phase argument.

Parameters

- **argument** (`Signal`) – The phase of the oscillator at each time sample.
- **midi_f0** (`Tensor`) – Fundamental frequency in midi.

Return type `Signal`

partials_constant(*midi_f0*)
Calculates a value to determine the number of overtones in the resulting square / saw wave, in order to keep aliasing at an acceptable level. Higher fundamental frequencies require fewer partials for a rich sound; lower-frequency sounds can safely have more partials without causing audible aliasing.

Parameters **midi_f0** – Fundamental frequency in midi.

12.2.6 FmVCO

class torchsynth.module.**FmVCO**(*synthconfig*, *device=None*, ***kwargs*)
Bases: `torchsynth.module.VCO`

Frequency modulation VCO. Takes a modulation signal as instantaneous frequency (in Hz) rather than as a midi value.

Typical modulation is calculated in pitch-space (midi). For FM to work, we have to change the order of calculations. Here the modulation depth is re-interpreted as the "modulation index" which is tied to the fundamental of the oscillator being modulated:

$$I = \Delta f / f_m$$

where I is the modulation index, Δf is the frequency deviation imparted by the modulation, and f_m is the modulation frequency, both in Hz.

make_control_as_frequency(*midi_f0*, *mod_signal*)
Creates a time-varying control signal in instantaneous frequency (Hz).

Parameters

- **midi_f0** (`Tensor`) – Fundamental frequency in midi.
- **mod_signal** – FM modulation signal (interpreted as modulation index).

Return type `Signal`

oscillator(*argument*, *midi_f0*)
A cosine oscillator. . . . Good ol' cosine.

Parameters

- **argument** (`Signal`) – The phase of the oscillator at each time sample.
- **midi_f0** (`Tensor`) – Fundamental frequency in midi (ignored in this VCO).

Return type `Signal`

output(*midi_f0*, *mod_signal*)

Parameters

- **`midi_f0`** (`Tensor`) – note value in midi
- **`mod_signal`** (*`Signal`*) – audio rate frequency modulation signal

Return type *`Signal`*

12.2.7 VCA

class `torchsynth.module.`**`VCA`**(*synthconfig*, *device=None*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

Voltage controlled amplifier.

The VCA shapes the amplitude of an audio input signal over time, as determined by a control signal. To shape control-rate signals, use *`torchsynth.module.ControlRateVCA`*.

`output`(*audio_in*, *control_in*)

Parameters

- **`audio`** – Audio input to shape with the VCA.
- **`amp_control`** – Time-varying amplitude modulation signal.

Return type *`Signal`*

12.2.8 VCO

class `torchsynth.module.`**`VCO`**(*synthconfig*, *device=None*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

Base class for voltage controlled oscillators.

Think of this as a VCO on a modular synthesizer. It has a base pitch (specified here as a midi value), and a pitch modulation depth. Its call accepts a modulation signal between [-1, 1]. An array of 0's returns a stationary audio signal at its base pitch.

Parameters

- **`synthconfig`** (*`SynthConfig`*) – An object containing synthesis settings that are shared across all modules, typically specified by *`Voice`*, or some other, possibly custom *`AbstractSynth`* subclass.
- **`phase`** – Initial oscillator phase.

Module Parameters

`tuning`

tuning adjustment for VCO in midi.

(**Min**: -24.0, **Max**: 24.0, **Curve**: 1, **Symmetric**: False)

`mod_depth`

depth of the pitch modulation in semitones.

(**Min**: -96.0, **Max**: 96.0, **Curve**: 0.2, **Symmetric**: True)

`initial_phase`

Initial phase for this oscillator.

(**Min**: -3.1415927410125732, **Max**: 3.1415927410125732, **Curve**: 1, **Symmetric**: False)

make_argument (*freq*)

Generates the phase argument to feed an oscillating function to generate an audio signal.

Parameters **freq** (*Signal*) – Time-varying instantaneous frequency in Hz.

Return type *Signal*

make_control_as_frequency (*midi_f0*, *mod_signal=None*)

Generates a time-varying control signal in frequency (Hz) from a midi fundamental pitch and pitch-modulation signal.

Parameters

- **midi_f0** (Tensor) – Fundamental pitch value in midi.
- **mod_signal** (Optional[*Signal*]) – Pitch modulation signal in midi.

Return type *Signal*

oscillator (*argument*, *midi_f0*)

This function accepts a phase argument and generates output audio. It is implemented by the child class.

Parameters

- **argument** (*Signal*) – The phase of the oscillator at each time sample.
- **midi_f0** (Tensor) – Fundamental frequency in midi.

Return type *Signal*

output (*midi_f0*, *mod_signal=None*)

Generates audio signal from modulation signal.

Parameters

- **midi_f0** (Tensor) – Fundamental of note in midi note value (0-127).
- **mod_signal** (Optional[*Signal*]) – Modulation signal to apply to the pitch.

Return type *Signal*

12.3 Control Rate Modules

Control rate modules produce signals that are used to modulate parameters of other modules. For performance these modules run at a reduced sampling rate.

12.3.1 ControlRateModule

class torchsynth.module.**ControlRateModule** (*synthconfig*, *device=None*, ***kwargs*)

Bases: *torchsynth.module.SynthModule*

An abstract base class for non-audio modules that adapts the functions of `SynthModule` to run at *control_rate*.

property control_buffer_size

Size of the module output in samples.

Return type Tensor

property control_rate

Control rate frequency in Hz.

Return type `Tensor`

output (**args*, ***kwargs*)
Performs the main action of `ControlRateModule`. Each child class should override this method.

Return type `Signal`

seconds_to_samples (*seconds*)
Convenience function to calculate the number of samples corresponding to given a time value and `control_rate`. Returns a possibly fractional value.

Parameters **seconds** (`Tensor`) – Time value in seconds.

Return type `Tensor`

to_buffer_size (*signal*)
Fixes the length of a signal to the control buffer size of this module, as specified by `control_buffer_size`. Longer signals are truncated to length; shorter signals are zero-padded.

Parameters **signal** (`Signal`) – A signal to pad or truncate.

Return type `Signal`

12.3.2 ADSR

class torchsynth.module.**ADSR** (*synthconfig*, *device=None*, ***kwargs*)
Bases: `torchsynth.module.ControlRateModule`

Envelope class for building a control-rate ADSR signal.

Parameters

- **synthconfig** (`SynthConfig`) – An object containing synthesis settings that are shared across all modules, typically specified by `Voice`, or some other, possibly custom `AbstractSynth` subclass.
- **device** (`Optional[device]`) – An object representing the device on which the *torch* tensors are allocated (as per PyTorch, broadly).

Module Parameters ADSR Parameters

attack
attack time (sec).

(**Min**: 0.0, **Max**: 2.0, **Curve**: 0.5, **Symmetric**: False)

decay
decay time (sec).

(**Min**: 0.0, **Max**: 2.0, **Curve**: 0.5, **Symmetric**: False)

sustain
sustain amplitude 0-1. The only part of ADSR that (confusingly, by convention) is not a time value..

(**Min**: 0.0, **Max**: 1.0, **Curve**: 1, **Symmetric**: False)

release
release time (sec).

(**Min**: 0.0, **Max**: 5.0, **Curve**: 0.5, **Symmetric**: False)

alpha
envelope curve. 1 is linear, >1 is exponential..

(**Min**: 0.1, **Max**: 6.0, **Curve**: 1, **Symmetric**: False)

make_attack(*attack_time*)
Builds the attack portion of the envelope.

Parameters **attack_time** – Length of the attack in seconds.

Return type *Signal*

make_decay(*attack_time*, *decay_time*)
Creates the decay portion of the envelope.

Parameters

- **attack_time** – Length of the attack in seconds.
- **decay_time** – Length of the decay time in seconds.

Return type *Signal*

make_release(*note_on_duration*)
Creates the release portion of the envelope.

Parameters **note_on_duration** – Duration of midi note in seconds (release starts when the midi note is released).

Return type *Signal*

output(*note_on_duration*)
Generate an ADSR envelope.

By default, this envelope reacts as if it was triggered with midi, for example playing a keyboard. Each midi event has a beginning and end: note-on, when you press the key down; and note-off, when you release the key. *note_on_duration* is the amount of time that the key is depressed.

During the note-on, the envelope moves through the attack and decay sections of the envelope. This leads to musically-intuitive, but programatically-counterintuitive behaviour:

Example

Assume attack is .5 seconds, and decay is .5 seconds. If a note is held for .75 seconds, the envelope won't pass through the entire attack-and-decay (specifically, it will execute the entire attack, and only .25 seconds of the decay).

If this is confusing, don't worry about it. ADSR's do a lot of work behind the scenes to make the playing experience feel natural.

Parameters **note_on_duration** (Tensor) – Duration of note on event in seconds.

Return type *Signal*

ramp(*duration*, *start=None*, *inverse=False*)
Makes a ramp of a given duration in seconds.

The construction of this matrix is rather cryptic. Essentially, this method works by tilting and clipping ramps between 0 and 1, then applying a scaling factor *alpha*.

Parameters

- **duration** (Tensor) – Length of the ramp in seconds.
- **start** (Optional[Tensor]) – Initial delay of ramp in seconds.

- **inverse** (Optional[bool]) – Toggle to flip the ramp from ascending to descending.

Return type *Signal*

12.3.3 ControlRateVCA

class torchsynth.module.**ControlRateVCA**(*synthconfig*, *device=None*, ***kwargs*)

Bases: *torchsynth.module.ControlRateModule*

Voltage controlled amplifier.

The VCA shapes the amplitude of a control input signal over time, as determined by another control signal. To shape audio-rate signals, use *torchsynth.module.VCA*.

output(*audio_in*, *control_in*)

Parameters

- **control** – Control signal input to shape with the VCA.
- **amp_control** – Time-varying amplitude modulation signal.

Return type *Signal*

12.3.4 LFO

class torchsynth.module.**LFO**(*synthconfig*, *exponent=tensor(2.7183)*, ***kwargs*)

Bases: *torchsynth.module.ControlRateModule*

Low Frequency Oscillator.

The LFO shape can be any mixture of sine, triangle, saw, reverse saw, and square waves. Contributions of each base-shape are determined by the `lfo_types` values, which are between 0 and 1.

Parameters

- **synthconfig** (*SynthConfig*) – See `SynthConfig`.
- **exponent** (Tensor) – A non-negative value that determines the discrimination of the soft-max selector for LFO shapes. Higher values will tend to favour one LFO shape over all others. Lower values will result in a more even blend of LFO shapes.

Module Parameters

frequency
Frequency in Hz of oscillation.

(**Min**: 0.0, **Max**: 20.0, **Curve**: 0.25, **Symmetric**: False)

mod_depth
LFO rate modulation in Hz.

(**Min**: -10.0, **Max**: 20.0, **Curve**: 0.5, **Symmetric**: True)

initial_phase
Initial phase of LFO.

(**Min**: -3.1415927410125732, **Max**: 3.1415927410125732, **Curve**: 1, **Symmetric**: False)

make_control(*mod_signal=None*)

Applies the LFO-rate modulation signal to the LFO base frequency.

Parameters **`mod_signal`** (`Optional`[*`Signal`*]) – Modulation signal in Hz. Positive values increase the LFO base rate; negative values decrease it.

Return type *`Signal`*

`make_lfo_shapes`(*argument*)

Generates five separate signals for each LFO shape and returns them as a tuple, to be mixed by *`torchsynth.module.LFO.output()`*.

Parameters **`argument`** (*`Signal`*) – Time-varying phase to generate LFO signals.

Return type `Tuple`[`Tensor`, `Tensor`, `Tensor`, `Tensor`, `Tensor`]

`output`(*mod_signal=None*)

Generates low frequency oscillator control signal.

Parameters **`mod_signal`** (`Optional`[*`Signal`*]) – LFO rate modulation signal in Hz. To modulate the depth of the LFO, use *`torchsynth.module.ControlRateVCA`*.

Return type *`Signal`*

12.3.5 ModulationMixer

`class` `torchsynth.module.`**`ModulationMixer`**(*synthconfig*, *n_input*, *n_output*, *curves=None*, *input_names=None*, *output_names=None*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

A modulation matrix that combines N input modulation signals to make M output modulation signals. Each output is a linear combination of all in input signals, as determined by an $NimesM$ mixing matrix.

Parameters

- **`synthconfig`** (*`SynthConfig`*) – See `SynthConfig`.
- **`n_input`** (`int`) – Number of input signals to module mix.
- **`n_output`** (`int`) – Number of output signals to generate.
- **`curves`** (`Optional`[`List`[`float`]]) – A positive value that determines the contribution of each input signal to the other signals. A low value discourages over-mixing.

`forward`(**signals*)

Performs mixture of modulation signals.

Return type `Tuple`[*`Signal`*]

12.4 Parameter Modules

Parameter modules simply output values.

12.4.1 CrossfadeKnob

class `torchsynth.module.`**`CrossfadeKnob`**(*synthconfig*, *device=None*, ***kwargs*)
Bases: *`torchsynth.module.SynthModule`*

Crossfade knob parameter with no signal generation

Module Parameters

`ratio`
crossfade knob.

(**Min**: 0.0, **Max**: 1.0, **Curve**: 1, **Symmetric**: False)

12.4.2 HardModeSelector

class `torchsynth.module.`**`HardModeSelector`**(*synthconfig*, *n_modes*, ***kwargs*)
Bases: *`torchsynth.module.SynthModule`*

A hard mode selector. NOTE: This is non-differentiable.

`forward`()
Wrapper for output that ensures a *`buffer_size`* length output.

Return type `Tuple`[`Tensor`, `Tensor`]

12.4.3 MonophonicKeyboard

class `torchsynth.module.`**`MonophonicKeyboard`**(*synthconfig*, *device=None*, ***kwargs*)
Bases: *`torchsynth.module.SynthModule`*

A keyboard controller module. Mimics a mono-synth keyboard and contains parameters that output a midi_f0 and note duration.

Module Parameters

`midi_f0`
pitch value in ‘midi’ (69 = 440Hz).

(**Min**: 0.0, **Max**: 127.0, **Curve**: 1.0, **Symmetric**: False)

`duration`
note-on button, in seconds.

(**Min**: 0.01, **Max**: 4.0, **Curve**: 0.5, **Symmetric**: False)

`forward`()
Wrapper for output that ensures a *`buffer_size`* length output.

Return type `Tuple`[`Tensor`, `Tensor`]

12.4.4 SoftModeSelector

class `torchsynth.module.`**`SoftModeSelector`**(*synthconfig*, *n_modes*, *exponent=tensor(2.7183)*, ***kwargs*)

Bases: *`torchsynth.module.SynthModule`*

A soft mode selector. If there are n different modes, return a probability distribution over them.

TODO: Would be nice to sample in a way that maximizes KL-divergence from uniform: https://github.com/torchsynth/torchsynth/issues/165

`forward`()

Normalize all mode weights so they sum to 1.0

Return type `Tuple`[`Tensor`, `Tensor`]

CHAPTER THIRTEEN

TORCHSYNTH.PARAMETER

13.1 ModuleParameter

class torchsynth.parameter.**ModuleParameter**(*value: Optional[torch.Tensor] = None*, *parameter_name: str = ''*, *parameter_range: Optional[torchsynth.parameter.ModuleParameterRange] = None*, *data: Optional[torch.Tensor] = None*, *requires_grad: bool = True*, *frozen: Optional[bool] = False*)

Bases: `torch.nn.parameter.Parameter`

ModuleParameter class that inherits from pytorch `Parameter`

TODO: Rethink value vs data here see https://github.com/torchsynth/torchsynth/issues/101

TODO: parameter_range shouldn't be optional see https://github.com/torchsynth/torchsynth/issues/340

Parameters

- **value** – initial value of this parameter in the human-readable range.
- **parameter_name** – A name for this parameter
- **parameter_range** – A *ModuleParameterRange* object that supports conversion between human-readable range and machine-readable [0,1] range.
- **data** – directly add data to this parameter in machine-readable range.
- **requires_grad** – whether or not a gradient is required for this parameter
- **frozen** – freeze parameter value and prevent updating

from_0to1()

Get the value of this parameter in the human-readable range.

TODO ModuleParameterRange should not be optional see https://github.com/torchsynth/torchsynth/issues/340 If no parameter range was specified, then the original parameter is returned.

Return type `Tensor`

static is_parameter_frozen(*parameter*)

Check whether a *ModuleParameter* is frozen. Asserts that parameter is an instance of *ModuleParameter*, and returns a bool indicating whether it is frozen.

Parameters parameter (*ModuleParameter*) – parameter to check

to_0to1(*new_value*)

Set the value of this parameter using an input that is in the human-readable range. Raises a runtime error if this parameter has been frozen.

Parameters **`new_value`** (`Tensor`) – new value to update this parameter with

13.2 ModuleParameterRange

class `torchsynth.parameter.`**`ModuleParameterRange`**(*minimum*, *maximum*, *curve=1*, *symmetric=False*, *name=None*, *description=None*)

Bases: `object`

ModuleParameterRange class is a structure for keeping track of the specific range that a parameter might take on. Also handles functionality for converting between machine-readable range [0, 1] and a human-readable range [minimum, maximum].

Parameters

- **`minimum`** (`float`) – minimum value in human-readable range
- **`maximum`** (`float`) – maximum value in human-readable range
- **`curve`** (`float`) – strictly positive shape of the curve values less than 1 place more emphasis on smaller values and values greater than 1 place more emphasis on larger values. 1 is linear.
- **`symmetric`** (`bool`) – whether or not the parameter range is symmetric, allows for curves around a center point. When this is True, a curve value of one is linear, greater than one emphasizes the minimum and maximum, and less than one emphasizes values closer to $(maximum - minimum)/2$.
- **`name`** (`Optional[str]`) – name of this parameter
- **`description`** (`Optional[str]`) – optional description of this parameter

`from_0to1`(*normalized*)

Set value of this parameter using a normalized value in the range [0,1]

Parameters **`normalized`** (`Tensor`) – value within machine-readable range [0, 1] to convert to human-readable range [minimum, maximum].

Return type `Tensor`

`to_0to1`(*value*)

Convert from human-readable range [minimum, maximum] to machine-range [0, 1].

Parameters **`value`** (`Tensor`) – value within the range defined by minimum and maximum

Return type `Tensor`

CHAPTER

FOURTEEN

TORCHSYNTH.SIGNAL

class `torchsynth.signal.`**Signal**

Bases: `torch.Tensor`

A convenience type for batched signals, either audio signals or control signals. A signal is 2D `Tensor`: *batch* x *num_samples*.

Note: To cast a tensor as a signal: `torch.zeros(batch_size, N).as_subclass(Signal)`

property batch_size

property num_samples

CHAPTER

FIFTEEN

TORCHSYNTH.SYNTH

SynthModule wired together form a modular synthesizer. *Voice* is our default synthesizer, and is used to generate synth1B1.

We base off pytorch-lightning `LightningModule` because it makes multi-GPU inference easy. Nonetheless, you can treat each synth as a native torch `Module`.

class torchsynth.synth.**AbstractSynth**(*synthconfig=None*, **args*, ***kwargs*)

Bases: `pytorch_lightning.core.lightning.LightningModule`

Base class for synthesizers that combine one or more *SynthModule* to create a full synth architecture.

Parameters **synthconfig** (`Optional`[*SynthConfig*]) – Global configuration for this synth and all its component *SynthModule*. If none is provided, we use our defaults.

add_synth_modules(*modules*)

Add a set of named children *SynthModule* to this synth. Registers them with the torch `Module` so that all parameters are recognized.

Parameters **modules** (`List`[`Tuple`[`str`, *SynthModule*, `Optional`[`Dict`[`str`, `Any`]]]]) – A list of *SynthModule* classes with their names and any parameters to pass to their constructor.

property batch_size

Return type `Tensor`

property buffer_size

Return type `Tensor`

property buffer_size_seconds

Return type `Tensor`

forward(*batch_idx=None*, **args*, ***kwargs*)

Wrapper around *output*, which optionally randomizes the synth *ModuleParameter* values in a deterministic way, and optionally disables gradient computations. This all depends on `synthconfig`.

Parameters **batch_idx** (`Optional`[`int`]) – If provided, we generate this batch, in a deterministic random way, according to `batch_size`. If None (default), we just use the current module parameter settings.

Return type `Tuple`[*Signal*, `Tensor`, `Optional`[`Tensor`]]

Returns

audio, parameters, is_train as a Tuple.

(batch_size x buffer_size audio tensor,

batch_size x n_parameters [0, 1] parameters tensor,

batch_size Boolean tensor of is this example train [or test], None if batch_idx is None)

freeze_parameters(*params*)
Freeze a set of parameters by passing in a tuple of the module and param name.

get_parameters(*include_frozen=False*)
Returns a dictionary of *ModuleParameterRange* for this synth, keyed on a tuple of the *SynthModule* name and the parameter name.

Parameters include_frozen (bool) – If some parameter is frozen, return it anyway.

Return type Dict[Tuple[str, str], *ModuleParameter*]

property hyperparameters
Returns a dictionary of curve and symmetry hyperparameter values keyed on a tuple of the module name, parameter name, and hyperparameter name

Return type Dict[Tuple[str, str, str], Any]

load_hyperparameters(*nebula*)
Load hyperparameters from a JSON file

Parameters nebula (str) – nebula to load. This can either be the name of a nebula that is included in torchsynth, or the filename of a nebula json file to load.

TODO add nebula list in docs See https://github.com/torchsynth/torchsynth/issues/324

Return type None

on_post_move_to_device()
LightningModule trigger after this Synth has been moved to a different device. Use this to update children SynthModules device settings

Return type None

output(**args, **kwargs*)
Each *AbstractSynth* should override this.

Return type *Signal*

randomize(*seed=None*)
Randomize all parameters

property sample_rate

Return type Tensor

save_hyperparameters(*filename, indent=True*)
Save hyperparameters to a JSON file

Return type None

set_hyperparameter(*hyperparameter, value*)
Set a hyperparameter. Pass in the module name, parameter name, and hyperparameter to set, and the value to set it to.

set_parameters(*params, freeze=False*)
Set various *ModuleParameter* for this synth.

Parameters

- **params** (Dict[Tuple[str, str], Tensor]) – Module and parameter strings, with the corresponding value.

- **`freeze`** (`Optional[bool]`) – Optionally, freeze these parameters to prevent further updates.

`test_step`(*batch*, *batch_idx*)
This is boilerplate required by pytorch-lightning `LightningTrainer` when calling test.

`training:` **`bool`**

`unfreeze_all_parameters`()
Unfreeze all parameters in this synth.

`class` `torchsynth.synth.`**`Voice`**(*synthconfig=None*, *nebula='default'*, **args*, ***kwargs*)
Bases: *`torchsynth.synth.AbstractSynth`*

The default configuration in torchsynth is the Voice, which is the architecture used in synth1B1. The Voice architecture comprises the following modules: a *`MonophonicKeyboard`*, two *`LFO`*, six *`ADSR`* envelopes (each *`LFO`* module includes two dedicated *`ADSR`*: one for rate modulation and another for amplitude modulation), one *`SineVCO`*, one *`SquareSawVCO`*, one *`Noise`* generator, *`VCA`*, a *`ModulationMixer`* and an *`AudioMixer`*. Modulation signals generated from control modules (*`ADSR`* and *`LFO`*) are upsampled to the audio sample rate before being passed to audio rate modules.

You can find a diagram of Voice in Synth Architectures documentation.

`output`()
Each *AbstractSynth* should override this.

Return type `Tensor`

`precision:` **`int`**

`training:` **`bool`**

`use_amp:` **`bool`**

CHAPTER SIXTEEN

TORCHSYNTH.UTIL

Utility functions for torch DSP related things

`torchsynth.util.`**`fix_length`**`(`*signal*, *length*`)`

Pad or truncate `Signal` to specified length.

Return type *`Signal`*

`torchsynth.util.`**`midi_to_hz`**`(`*midi*`)`

Convert from midi (linear pitch) to frequency in Hz.

Parameters **`midi`** (`Tensor`) – Linear pitch on the MIDI scale.

Return type `Tensor`

Returns Frequency in Hz.

`torchsynth.util.`**`normalize`**`(`*signal*`)`

Normalize every individual signal in batch.

Return type *`Signal`*

`torchsynth.util.`**`normalize_if_clipping`**`(`*signal*`)`

Only normalize invidiaul signals in batch that have samples less than -1.0 or greater than 1.0

Return type *`Signal`*

CHAPTER SEVENTEEN

INDEX

CHAPTER

EIGHTEEN

SEARCH PAGE

PYTHON MODULE INDEX

INDEX