pyfxr

Release 0.3.0

Daniel Pope

Apr 18, 2021

CONTENTS:

pyfxr generates tones and noises in fast Cython code, and is intended for use in simple Python computer games and in education. It can generate:

- Highly configurable noises (the original sfxr)
- Pure tones with sine, square, saw and triangle waveforms
- Pluck sounds, like harp or guitar, using the Karplus-Strong algorithm

Sounds can be played with any library that supports the buffer protocol (such as Pygame), or saved to `.wav` files.

For example, this is a complete program to generate a 1s pluck sound and play it with Pygame:

```
import pygame.mixer
import time
import pyfxr

# pyfxr generates mono 44kHz sounds so we must set
# Pygame to use this
pygame.mixer.pre_init(44100, channels=1)
pygame.mixer.init()

tone = pyfxr.pluck(duration=1.0, pitch='A4')
pygame.mixer.Sound(buffer=tone).play()

# wait for the sound to finish before exiting
time.sleep(tone.duration)
```

CHAPTER

ONE

GENERATING SOUNDS

pyfxr has 3 sound generation algorithms, described below.

1.1 sfxr-style sounds

sfxr is a user interface for generating sounds with a wide array of parameters. pyfxr provides a full API to generate these sounds in Python programs.

class pyfxr.**SFX**(***kwargs*)

Build a sound effect using a set of parameters.

The list of parameters is long and the sensible ranges for the parameters aren't that clear. This class acts as a validator and builder for the parameters, making it simpler to experiment with sound effects.

You can also serialise this class in several ways:

- The `repr()` is suitable for pasting into code.
- You can serialise it as JSON using `.as_dict()`.
- You can pickle the class.

In any of these case the size is much smaller than the generated SoundBuffer.

SFX supports the buffer protocol much like *SoundBuffer*; accessing the object as a buffer generates and caches a sound.

base_freq: float
The initial frequency of the sound

freq_limit: float
The minimum frequency of the sound

freq_ramp: float
The rate of change of the frequency of the sound

freq_dramp: float
The acceleration of the change in frequency of the sound

duty: float
If using square wave, the duty cycle of the waveform

duty_ramp: float
The rate of change of the square wave duty cycle

vib_strength: float
Vibrato strength

vib_speed: float
Vibrato speed

vib_delay: float
Vibrato delay

env_attack: float
The duration of the attack phase of the ADSR envelope

env_sustain: float
The duration of the sustain phase of the ADSR envelope

env_decay: float
The duration of the decay phase of the ADSR envelope

env_punch: float
Causes the volume to decrease during the sustain phase of the envelope

lpf_resonance: float
Low-pass filter resonance

lpf_freq: float
Low-pass filter cutoff frequency

lpf_ramp: float
Low-pass filter cutoff ramp

hpf_freq: float
High-pass filter frequency

hpf_ramp: float
High-pass filter ramp

pha_offset: float
Phaser offset

pha_ramp: float
Phaser ramp

repeat_speed: float
Repeat speed

arp_speed: float
Arpeggio speed

arp_mod: float
Arpeggio mod

property wave_type
Get the wave type.

as_dict() → dict
Get the parameters as a dict.

The dict is suitable for serialising as JSON; to reconstruct the object, pass the parameters as kwargs to the constructor, eg.

```
>>> s = SFX(...)
>>> params = s.as_dict()
>>> s2 = SFX(**params)
```

build() → _pyfxr.SoundBuffer
Get the generated sound (memoised).

envelope(*attack: float = 0.0*, *sustain: float = 0.3*, *decay: float = 0.4*, *punch: float = 0.0*)
Set the ADSR envelope for this sound effect.

The `wave_type` of an SFX must be one of these values:

class pyfxr.**WaveType**(*value*)
The wave types available for the SFX builder.

Pure tones with tone() use arbitrary wavetables rather than this enumeration.

SQUARE = 0
A square-wave waveform

SAW = 1
A saw-wave waveform

SINE = 2
A sine wave

NOISE = 3
Random noise

You can also randomly generate those parameters:

pyfxr.**pickup**() → *pyfxr.SFX*
Generate a random bell sound, like picking up a coin.

pyfxr.**laser**() → *pyfxr.SFX*
Generate a random laser sound.

pyfxr.**explosion**() → *pyfxr.SFX*
Generate a random explosion sound.

pyfxr.**powerup**() → *pyfxr.SFX*
Generate a random chime, like receiving a power-up.

pyfxr.**hurt**() → *pyfxr.SFX*
Generate a random impact sound, like a character being hurt.

pyfxr.**jump**() → *pyfxr.SFX*
Generate a random jump sound.

pyfxr.**select**() → *pyfxr.SFX*
Generate a random 'blip' noise, like selecting an option in a menu.

1.2 Wavetable sounds

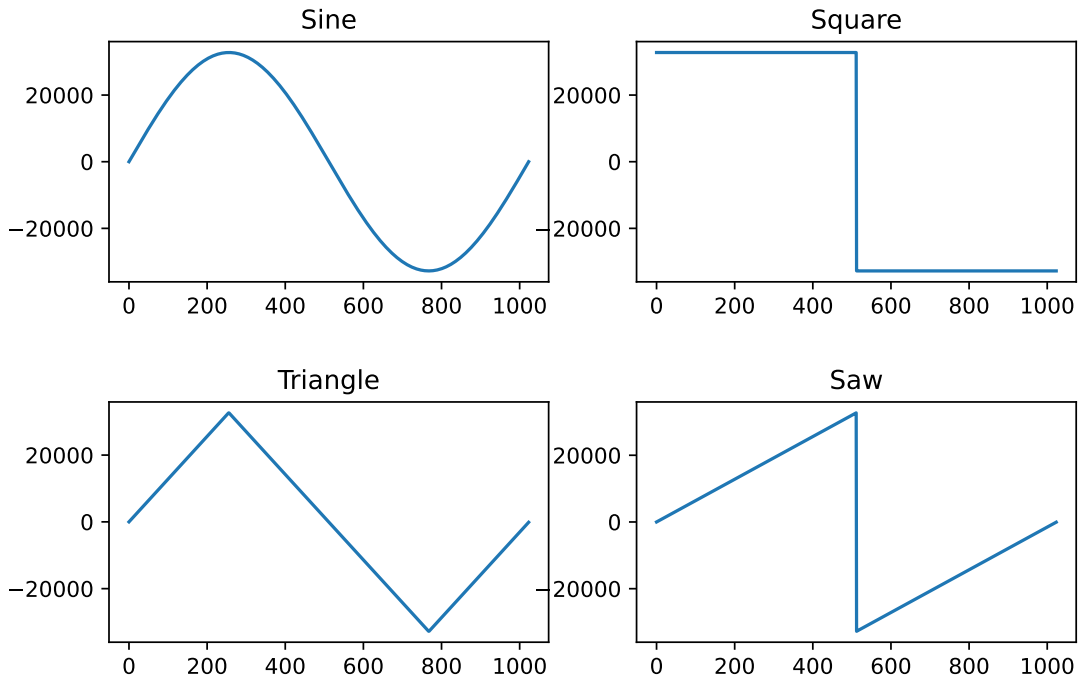
pyfxr can also generate pure tones using a wavetable. A wavetable gives the shape of a waveform, such as these:

Wavetables can have any shape. To construct a Wavetable with a custom shape, pass an iterable to the constructor. This should return 1024 float values in [-1, 1].

```
from math import pi, sin
from pyfxr import Wavetable

def gen():
    for i in range(1024):
        t = pi / 512 * i
        yield 0.75 * sin(t) + 0.25 * sin(3 * t + 0.5)

wt = Wavetable(gen())
```



Or perhaps more simply, use `Wavetable.from_function()`:

```
Wavetable.from_function(
    lambda t: 0.75 * sin(t) + 0.25 * sin(3 * t + 0.5)
)
```

class pyfxr.**Wavetable**(*gen*)

static from_function(*f*)
Generate a wavetable by calling a function f.

f should take a single float argument between 0 and tau (pi * 2) and return values in [-1, 1].

static saw()
Construct a saw waveform.

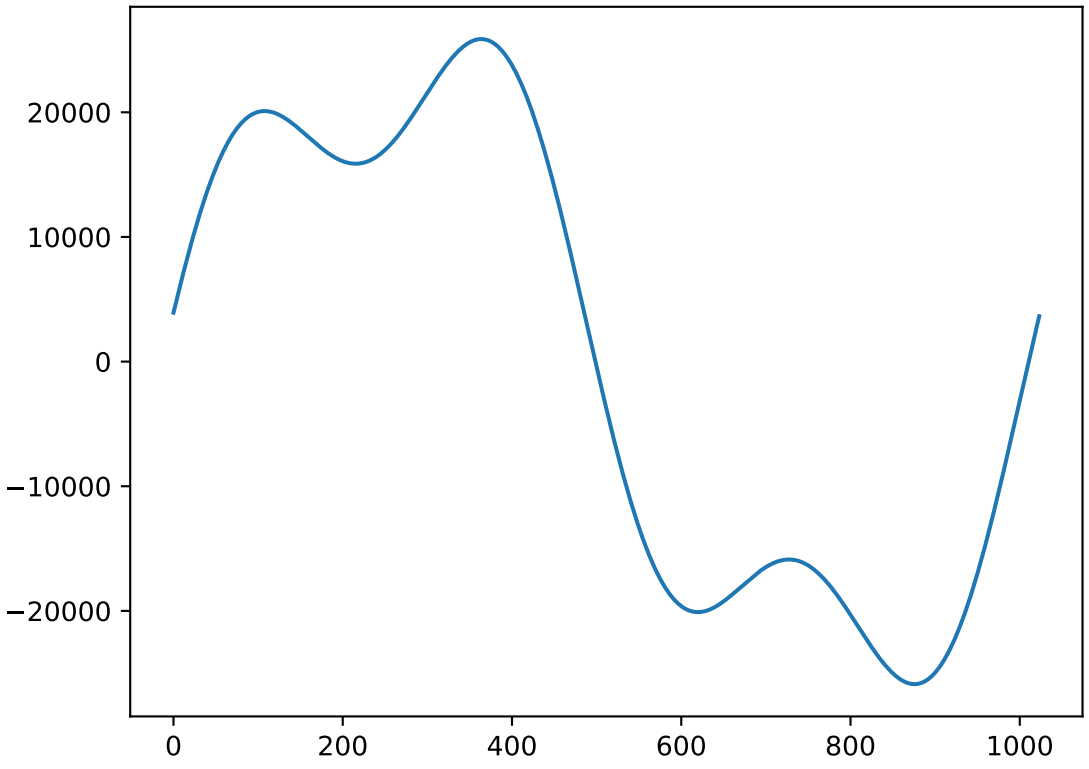
static sine()
Construct a sine waveform.

static square(*float duty_cycle=0.5*)
Generate a square-wave waveform.

duty_cycle is the fraction of the period during which the waveform is greater than zero.

static triangle()
Construct a triangle waveform.

pyfxr.**tone**(*pitch: Union[float, str] = 440.0*, *attack: float = 0.1*, *decay: float = 0.1*, *sustain: float = 0.75*, *release: float = 0.25*, *wavetable: _pyfxr.Wavetable = <_pyfxr.Wavetable object>*) → _pyfxr.SoundBuffer
Generate a tone using a wavetable.


20000
10000
0
−10000
−20000
0
200
400
600
800
1000

The tone will be modulated by an ADSR envelope (attack-decay-sustain-release) which gives the tone a more natural feel, and avoids clicks when played. The total length of the tone is the sum of these durations.

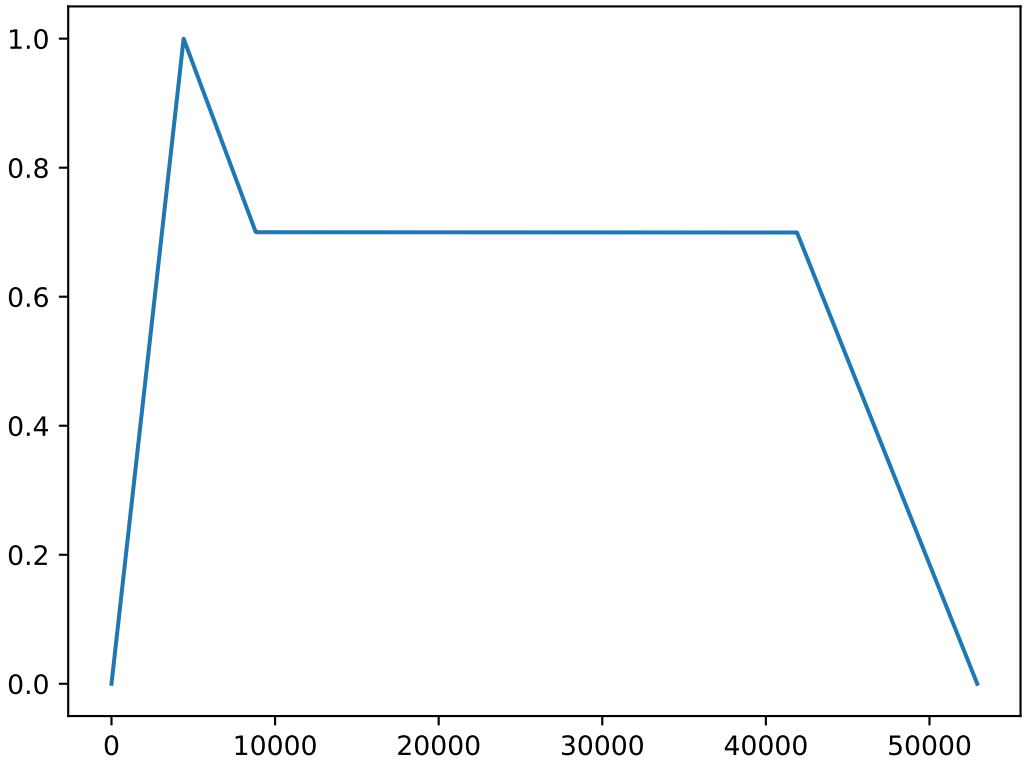
Parameters

- **wavetable** – The wavetable to use (default is a sine wave).
- **pitch** – The pitch of the tone to generate, either float Hz or a note name/number like `Bb4` for B-flat in the 4th octave.
- **attack** – Attack time in seconds
- **decay** – Decay time in seconds
- **sustain** – Sustain time in seconds
- **release** – Release time in seconds

1.2.1 ADSR Envelopes

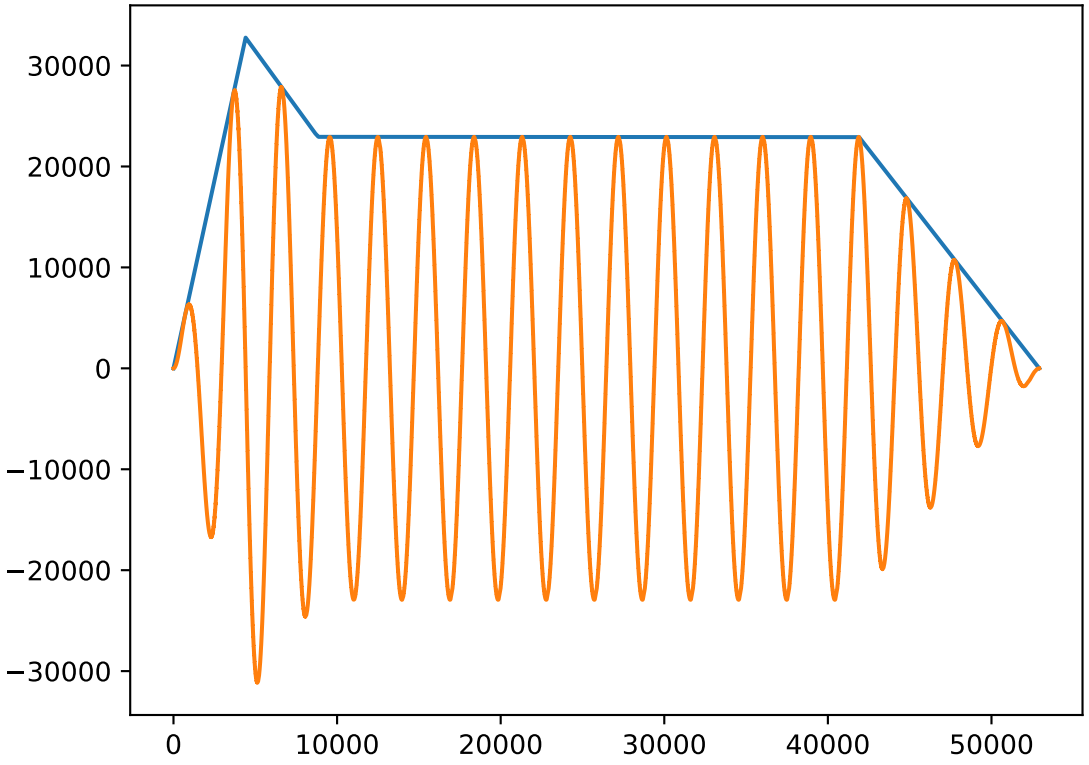
Tones are bounded by a 4-phase "ADSR Envelope". The phases are:

- **Attack** - initial increase in volume
- **Decay** - volume decreases to the sustain level
- **Sustain** - the volume stays constant while the note is held
- **Release** - the volume fades to zero



The default ADSR envelope has this shape. Note that durations for any of the ADSR phases can be set to zero to omit that phase. It is recommended to skip only decay and sustain phases, as attack and release phases help to avoid clicks when the sound plays.

This is applied to a waveform by multiplication:



1.3 Pluck sounds

pyfxr can also generate pluck sounds, like a guitar or harp.

`pyfxr.`**`pluck`**`()`

pluck(float duration, float pitch, float release=0.1) Generate a pluck sound using the Karplus-Strong algorithm.

CHAPTER

TWO

COMPOSING TOOLS

pyfxr.**chord**(*sounds: u'List[Union[SoundBuffer, SFX]]'*, *double stagger=0.0*) → *SoundBuffer*

Generate a chord by combining several sounds.

If stagger is given, the start of each additional sound will be delayed by *stagger* seconds.

pyfxr.**simple_chord**(*name: str*, *attack: float = 0.1*, *decay: float = 0.1*, *sustain: float = 0.75*, *release: float = 0.25*, *wavetable: _pyfxr.Wavetable = <_pyfxr.Wavetable object>*, *stagger: float = 0.0*) → _pyfxr.SoundBuffer

Construct a chord using a chord name like

- *C* - major chord in C
- *Bbm* or *Bb-* - minor chord in B-flat
- *D7* - dominant 7th

etc.

Other parameters are as for `tone()` and *:func:`chord*.

CHAPTER

THREE

USING SOUNDBUFFER OBJECTS

pyfxr's *sound generation APIs* return `SoundBuffer` and `SFX` objects.

A soundbuffer is a packed sequence of 16-bit samples:

```
>>> buf = pyfxr.explosion().build()
>>> len(buf)
32767
>>> buf[0]
2418
```

but more importantly it supports the buffer protocol, which allows it to be passed directly to many sound playing APIs (see below).

You can also save a SoundBuffer to a `.wav` file, which is very widely supported:

```
buf.save("explosion1.wav")
```

An SFX object is a set of parameters to generate a SoundBuffer. You can generate and retrieve the SoundBuffer with `SFX.build()`, but you can also play an SFX just like a SoundBuffer.

class pyfxr.**SoundBuffer**

sample_rate: int
The sample rate in samples per second. Currently, always 44100.

channels: int
The number of channels in the sample. Currently, always 1 (mono).

duration
Get the duration of this sound in seconds, as a float.

get_queue_source(*self*)
Duck type as a pyglet.media.Source.

save(*self*, *unicode filename: str*)
Save this sound to a .wav file.

3.1 With Pygame

Pygame can construct a sound from any buffer object, including SoundBuffer:

```
buf = pyfxr.tone()
pygame.mixer.Sound(buffer=buf)
```

Be aware that as of Pygame 2.0.1, `Sound` objects do not have their own sample rate and mono/stereo information; they are assumed to have the same format as the mixer. For correct playback you must initialise the mixer to 44100 kHz mono:

```
pygame.mixer.pre_init(pyfxr.SAMPLE_RATE, channels=1)
pygame.mixer.init()
```

3.2 With Pyglet

SoundBuffers can also be used as Pyglet media sources:

```
pyglet.media.StaticSource(buf)
```

This does not work by the buffer protocol; SoundBuffer has special adapter code to allow it to work like this.

3.3 With sounddevice

sounddevice provides access to sound devices, without being coupled to a game or UI framework.

`sounddevice` also supports the buffer protocol and can play SoundBuffers directly:

```
import sounddevice
import pyfxr

sounddevice.play(pyfxr.jump(), pyfxr.SAMPLE_RATE)
```

CHAPTER

FOUR

CHANGES

4.1 0.3.0

- New: GUI to explore *SFX* parameters
- New: some parameters for SFX require positive numbers
- New: *chord()* for combining sounds
- New: *simple_chord()* for generating harmonic chords from chord names

4.2 0.2.0

- New: *SFX*, which manages parameters for sfx generation
- Change: `jump()`, `explosion()` etc now return `SFX` instances.
- New: GUI now prints code for sounds generated
- Deprecation: `sfx()` is now deprecated, use *SFX*.
- Fix: bug in handling of `arp_mod` parameter

4.3 0.1.0

Initial release.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

W