Szoftverrendszerek tesztelése
-projekt-

Készítette: Horváth Levente

Tartalomjegyzék

1. Bevezető

A pytest egy hatékony és könnyen használható tesztelési keretrendszer Pythonban, amely segít a fejlesztőknek automatizálni és szervezni a tesztjeiket. A pytest segít az automatizált tesztelés során az előre meghatározott eredmények ellenőrzésében, így javítva a fejlesztési folyamat minőségét és hatékonyságát. A pytest tesztelési keretrendszer által biztosított rugalmasság lehetővé teszi a tesztek egyszerű kezelését, hibakeresést és javítást. A pytest előnye, hogy magasabb szintű absztrakciókat és nagyobb rugalmasságot biztosít, így a tesztelési kód olvashatóbb és karbantarthatóbb lesz[1].

A tesztelést az államvizsga témám köré építettem fel, mely digitális jelfeldolgozásra (Digital Signal Processing/DSP) összpontosít. A digitális jelfeldolgozás egy olyan technika, amelyet diszkrét jelek, például hang, kép és adat feldolgozására és elemzésére használnak. Az így feldolgozott digitális jelek számsorozatok, amelyek egy folytonos változó mintáit reprezentálják egy olyan tartományban, mint az idő, a tér vagy a frekvencia. A DSP alkalmazásai közé tartozik többek között a hang- és beszédfeldolgozás, képfeldolgozás, adattömörítés, videó- és hangkódolás, biológiai jelek feldolgozása és a szeizmológia is[2].

A projektem célja, hogy egy vagy több előre mintavételezett nyers gitár hangállományt használva különböző effektusokat hozzak létre Python programozási nyelv alatt, ezért ezen félév során egységteszteket, azaz unit testeket használtam, hogy miközben haladok, tudjam ellenőrizni a kódokat, amiket írok.

2. A projekt felépítése

Sok féle gitár effektus van már, azonban sok közülük multieffekt processzorokon van illetve gitárpedálokon, ezeknek pedig nagyon borsos áruk van. Célom az volt, hogy saját programmal, saját effektusokat hozzak létre és hasonló hangzást érjek el. Munkám során az alábbi effektusok megvalósításával és tanulmányozásával foglalkoztam:

- Torzítás
- Overdrive
- Késleltetés és konvolúciós téreffektus
- EQ

Ezeknek külön osztályokat hoztam létre Pythonban egy **effects** könyvtárban, amelyeket be is mutatok sorra, majd a végén azt, hogy hogyan kell mindegyiket meghívni a **runnable/main.py** állományban.

Torzítás

```python
import numpy as np
from utils.utils import clip


class Distortion:
    def __init__(self, gain: float = 1.0):
        self.gain = gain

    def apply(self, data_in: float) -> float:
        applied_distortion = np.sign(data_in) * (1.0 - np.exp(self.gain * np.sign(data_in) * data_in))
        return clip(applied_distortion)
```

A torzítás amit használok, egy egyszerű hatást valósít meg: gondoljunk arra, hogy egy hangfalnak adunk jelet. Ha az erősítés mértéke alacsony, akkor a hangfal szépen és tisztán fogja megszólaltatni az adott jelet. Azonban ha az erősítés túl magasra van állítva, akkor az erősítő nem tudja megfelelően kezelni a jelet, és a hangfalba küldött jel torzulni fog.

Az 1. képlet lényegében egy új jelet hoz létre, amely a bemeneti jel torzított változata. Legyen a bemeneti jel x[n], az EQ-nál is használt gain paraméter a nemlinearitás mértékének szabályozásához, illetve a kimeneti jel y[n]. Ekkor a torzított jel y[n] a következőképpen áll elő:

$$y[n] = sign(x[n]) \cdot \left[1 - e^{gain \cdot sign(x[n])} \cdot sign(x[n])\right] (1)$$

ahol **sign(x)** a bemeneti jel előjele, vagyis $sign(x) = \{-1, ha\ x < 0; 0, ha\ x = 0; 1, ha\ x > 0\}$.

az így kapott kifejezés a bemeneti jel torzított változatát adja vissza. A gain paraméter meghatározza a nemlinearitás mértékét.

A **clip** függvény során használt normalizált -1 és 1 közötti tartomány használata fontos a digitális jelfeldolgozásban, mivel a jelek amplitúdója nem lehet nagyobb vagy kisebb bizonyos értékeknél, mivel azokat a digitális jelfeldolgozó eszközök nem tudják megfelelően kezelni.

Overdrive

```python
class Overdrive:
    def __init__(self, amplitude: float = 1.0, gain: float = 1.0):
        self.amplitude = amplitude
        self.gain = gain

    def apply_hard_clipping(self, data_in: float) -> float:
        applied_hard_clip = self.gain * data_in
        return clip(applied_hard_clip)

    def apply_soft_clipping_1(self, data_in: float) -> float:
        applied_soft_clip = self.amplitude * np.tanh(self.gain * data_in)
        return clip(applied_soft_clip)

    def apply_soft_clipping_2(self, data_in: float) -> float:
        applied_soft_clip = self.amplitude * np.arctan(self.gain * data_in)
        return clip(applied_soft_clip)
```

A torzításhoz hasonlóan az overdrive hatása is hasonló, viszont kontrolláltabb. Egy overdrive effektus esetében lehetőségünk van arra, hogy a torzítási szintet a saját igényeinkre szabjuk, és így a torzult hangzást még mindig kellemesnek és zenének tűnőnek halljuk [3].

3 fajta overdriveot valósítok meg: hard clippingből egyet és soft clippingből kettőt.

- Hard clipping
    - Az overdrive ezen típusa a kimeneti szint korlátozásával működik, amihez a bemeneti jelet egy erősítési tényezővel szorozzuk, majd normalizáljuk -1 és 1 között.

$$y[n] = \frac{gain \cdot x[n]}{\max(|x[n]|)} \ (2)$$

- Soft clipping (1)
    - Ezen típusú overdrive esetében a bemeneti jelre egy nem lineáris függvényt alkalmazunk, amely "puhítja" az erősített és levágott jelek közötti átmenetet. Az egyenletben alkalmazott függvény a

hiperbolikus tangens (tanh), amelynek sima és fokozatos az átmenete a lineáris és a nem lineáris tartomány között.

$$y[n] = \frac{amplitude \cdot tanh(gain \cdot x[n])}{max(|tanh(gain \cdot x[n])|)} \quad (3)$$

- Soft clipping (2)
  - Ez az overdrive hasonló az előző soft clipping típushoz, de egy másik nem lineáris függvényt használ a kimeneti jel formálásához. Az egyenletben alkalmazott függvény az arkusz tangens (arctan). Az arctan hasonlóan sima az átmenetben a lineáris és nem lineáris tartományok között, de a görbe alakja eltér a tanh-tól.

$$y[n] = \frac{amplitude \cdot arctan(gain \cdot x[n])}{max(|arctan(gain \cdot x[n])|)} \quad (4)$$

Késleltetés/konvolúciós téreffektus

```python
class Delay:
    def __init__(self, fs, duration_sec: float = 0.5, amplitude: float = 0.8):
        self.fs = fs
        self.duration_sec = duration_sec
        self.amplitude = amplitude

    def apply_convolutional_delay(self, signal):
        delay_len_samples = round(self.duration_sec * self.fs)
        impulse_response = np.zeros(delay_len_samples)
        impulse_response[0] = 1
        impulse_response[-1] = self.amplitude
        output_sig = (np.convolve(signal, impulse_response) * 2 ** 15).astype(np.int16)
        return output_sig
```

A konvolúciós téreffektus az impulzusválaszt adja hozzá a késleltetett jelhez a megfelelő amplitúdóval. Ez az impulzusválasz egy olyan jellemző, amely leírja, hogy a késleltetett jel hogyan viselkedik az idő függvényében. Ezt követően a két jel összeszorzódik, és a kimeneti jel a konvolúció eredménye lesz.

$$y[n] = (x[n] * h[n]) \cdot 2^{15} \quad (5)$$

EQ (hangszín szabályozó)

Az EQ tervezésének egyik legfontosabb lépése hogy nekünk megfelelő szűrőtípust válasszunk. Ebben az esetben egy negyedrendű, 3 sávos Butterworth EQ-ra gondoltam. Ez maximálisan lapos frekvenciaválasszal rendelkezik az átviteli sávban, és amelyet a hangtechnikában gyakran használnak. Az IIR-szűrők más típusai, például a Chebyshev- és az elliptikus szűrők szintén gyakran használatosak EQ-khoz, és ezeknek is megvannak a maguk előnyei és hátrányai.

```python
from scipy.signal import butter, lfilter

class ButterworthEQ:
    def __init__(self, fs, low_cutoff, high_cutoff, low_gain, mid_gain, high_gain):
        self.fs = fs
        self.low_b, self.low_a = butter(4, low_cutoff / fs, 'low')
        self.high_b, self.high_a = butter(4, high_cutoff / fs, 'high')
        self.mid_b, self.mid_a = butter(4, [low_cutoff / fs, high_cutoff / fs], 'bandpass')
        self.low_gain = low_gain
        self.mid_gain = mid_gain
        self.high_gain = high_gain

    def apply(self, signal):
        low_signal = lfilter(self.low_b, self.low_a, signal)
        mid_signal = lfilter(self.mid_b, self.mid_a, signal)
        high_signal = lfilter(self.high_b, self.high_a, signal)
        filtered_signal = self.low_gain * low_signal + self.mid_gain * mid_signal +
self.high_gain * high_signal
        return filtered_signal
```

A Butterworth EQ-nak amit választottam az a lényege, hogy 3 különböző helyen tudja szűrni a bemeneti jelet. Van egy alul-, közép- és felüláteresztő szűrője, mindegyiknek külön meg lehet adni a fokszámát és a frekvenciát, ahol vágni szeretnénk. Ha a jel 3 pontján megtörtént a vágás, akkor ezt a 3 komponenst össze kell adni, és megkapjuk az EQ által szűrt jelet. Azt is megadhatjuk, hogy a bizonyos komponensek milyen erősítéssel rendelkezzenek, így kiválasztva, hogy a mély, közepes vagy magas hangokat szeretnénk jobban hallani.

AudioProcessor

Ez már nem effektus, hanem egy állományba írt segédosztály, amely lehetővé teszi az eddigi effektusok könnyebb használatát. A **main.py** állományban kell inicializálni.

```python
class AudioProcessor:
    def __init__(self, input_file_path, output_file_path):
        self.input_file_path = input_file_path
        self.output_file_path = output_file_path
        self.fs, self.signal_int16 = wavfile.read(self.input_file_path)
        self.signal_float64 = self.signal_int16.astype(float) / 2 ** 15

    def write_output_file(self, signal, file_name):
        file_path = os.path.join(self.output_file_path, file_name)
        wavfile.write(file_path, self.fs, signal)

    def apply_effect_and_save(self, effect_func):
        class_name = effect_func.__self__.__class__.__name__
        effect_name = f"{effect_func.__name__}_{class_name}"
        output_file_name = effect_name.lower() + ".wav"
        signal = effect_func(self.signal_float64) * 2 ** 15
        self.write_output_file(signal.astype('int16'), output_file_name)
```

Működésének lényege, hogy a kezeli a fájl beolvasását illetve az effektusok elmentését és írását.

3. Tesztelés és futtatható állományok működése

Tesztelés

A teszteléshez készítettem egy külön **Tests** osztályt, mely a **testing/tests.py-ban** található, amely számos tesztmódszert tartalmaz a különböző osztályok funkcionalitásának tesztelésére: **Distortion**, **Overdrive**, **Delay** és **ButterworthEQ**. Minden tesztmódszer létrehozza a tesztelt osztály egy példányát, és azt alkalmazza néhány bemeneti adatra. A kimenetet ezután az **assert** utasítás és a pytest modulból származó **approx** metódus segítségével összehasonlítjuk egy várható értékkel. A tesztelés a következőképpen néz ki:

```
class Tests:
    def test_distortion(self):
        # Test case 1
        distortion = Distortion(gain=2.0)
        data_in = 0.5
        distorted_data = distortion.apply(data_in)
        assert distorted_data == pytest.approx(-1.0)

        # Test case 2
        distortion = Distortion(gain=1.0)
        data_in = 0.5
        distorted_data = distortion.apply(data_in)
        assert distorted_data == pytest.approx(-0.64, abs=0.01)

    def test_overdrive(self):
        overdrive = Overdrive(amplitude=2.0, gain=3.0)
        data_in = 0.5
        assert overdrive.apply_hard_clipping(data_in) == pytest.approx(1.0)
        assert overdrive.apply_soft_clipping_1(data_in) == pytest.approx(1.098, abs=0.1)
        assert overdrive.apply_soft_clipping_2(data_in) == pytest.approx(1.057, abs=0.1)

    def test_convolutional_delay(self):
        fs = 8000
        duration_sec = 0.5
        amplitude = 0.8
        delay = Delay(fs, duration_sec, amplitude)
        signal = np.array([1, 2, 3])
        impulse_response = np.zeros(round(duration_sec * fs))
        impulse_response[0] = 1
        impulse_response[-1] = amplitude
        expected_output = (np.convolve(signal, impulse_response) * 2 ** 15).astype(np.int16)
        output = delay.apply_convolutional_delay(signal)
        assert np.allclose(output, expected_output)

    def test_butterworthEQ(self):
        fs = 8000
        low_cutoff = 500
        high_cutoff = 2000
        low_gain = 0.8
        mid_gain = 1.2
        high_gain = 1.5
```

```
        eq = ButterworthEQ(fs, low_cutoff, high_cutoff, low_gain, mid_gain, high_gain)
        signal = np.random.randn(fs)
        output = eq.apply(signal)

        # Verify that the output has the same shape as the input signal
        assert output.shape == signal.shape
```

Az itt megírt tesztek letesztelhetőek egy terminal ablakból, azonban a főfüggvényben is, mielőtt elvégeznénk a módosításokat a kívánt hangfájlokon.

Futtatás

Egy futtatható állományom van ez nem más mint a **runnable/main.py**, mely a következőképpen néz ki:

```
import os

from effects.audioprocessor import AudioProcessor
from effects.overdrive import Overdrive
from effects.distortion import Distortion
from effects.delay import Delay
from effects.butterworth_eq import ButterworthEQ

from testing.tests import Tests

if __name__ == "__main__":
    tests = Tests()
    tests.test_distortion()
    tests.test_overdrive
    tests.test_convolutional_delay()
    tests.test_butterworthEQ()

    input_file_path = os.path.abspath("./audio_inputs/gitar.wav")
    output_file_path = os.path.abspath("./audio_outputs/")

    processor = AudioProcessor(input_file_path, output_file_path)

    overdrive = Overdrive(amplitude=0.9, gain=5.0)
    distortion = Distortion(gain=5.0)
    delay = Delay(processor.fs, duration_sec=0.5, amplitude=0.8)
    butterworth_eq = ButterworthEQ(processor.fs, 500, 10000, 0.6, 1.4, 0.9)

    processor.apply_effect_and_save(distortion.apply)
    processor.apply_effect_and_save(overdrive.apply_soft_clipping_1)
    processor.apply_effect_and_save(delay.apply_convolutional_delay)
    processor.apply_effect_and_save(butterworth_eq.apply)

#run it like this: python -m runnable.main
```

Itt az történik, hogy minden szükséges állományt beimportolok, ezek után letesztelem a működésüket és ha minden helyes, továbbhalad a program a következő fázisra, melyben a felhasználó megadhatja a fájl vagy fájlok elérési útját,

illetve azt, hogy hova szeretné elmenteni a módosításokat. Ezután inicializál egy **AudioProcessor** típusú változót, melynek megadja az előbbi két elérési útvonalat. Ezután az effektusokat is inicializálja, mindegyiket a saját megfelelő osztályával, aztán az **AudioProcessor** típusú változón végrehajthatja ezeket az **apply_effect_and_save** metódussal.

4. Következtetés és továbbfejlesztés

A dolgozat célja az effektusok és a tesztelés bemutatása volt, az eredmények alapján megállapíthatjuk, hogy sikerült egy működőképes, saját fejlesztésű virtuális multieffekt processzort létrehoznom, mely képes a kívánt effektusok létrehozására és megvalósítására.

Az effektusok továbbfejlesztése számos lehetőséget kínál, egyik az lehet, hogy a meglévő effektusokat finomhangoljuk, hogy még jobb minőségű hangzást érjünk el. Ezen kívül új effektusok fejlesztése is lehetséges, hogy további kreatív lehetőségeket nyújtsunk a felhasználók számára, ez pedig nagyon egyszerű, hiszen csak egy új osztályt kell létrehoznunk új metódusokkal, és az **AudioProcessor**-nak hála nagyon gyorsan üzembe tudjuk helyezni.

5. Könyvészet

[1] https://docs.pytest.org/en/latest/

[2] Oppenheim, Alan V. and Cram. “Discrete-time signal processing” 3rd edition. (2011).

[3] https://jetpedals.com/blogs/news/clipping-diodes-and-how-they-affect-your-tone