Sound processing

Task 2

Subtractive synthesis of sound

Jakub Janaszkiewicz

1. Table of contents

Table of contents

2. Introduction

The aim of the exercise is to implement sound synthesis methods enabling to generate sound in realtime. These methods should be used in a self-constructed application for creating sound and sound effects. The user of the application should be able to modify the parameters of the algorithms and to immediately listen to the results.

3. Creating sound timbre and implementing the algorithm for its synthesis

3.1 Theoretical introduction

Creating the sound data was conducted with use of the algorithm from Figure 1. Different waves generated by all types of generators was summed up. The outcome from that process was next putted into the lowpass filter. The final result consist the filtered sound wave.

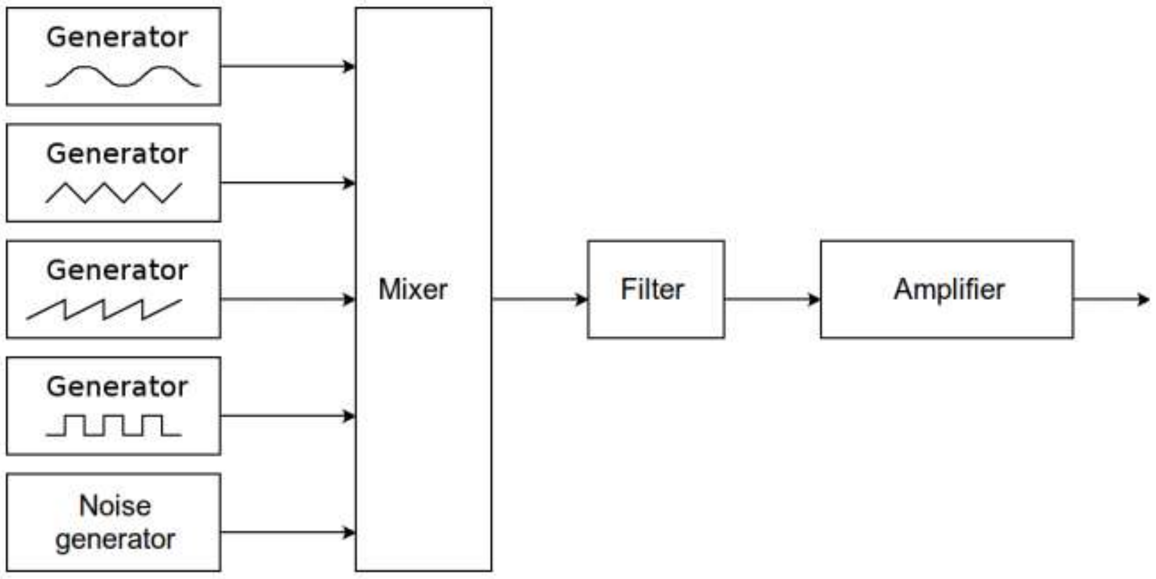


Figure 1 Algorithm used to generate sound data

The filter parameters were implemented as follows:

$s = \sin(2\pi f / f_s)$
$c = \cos(2\pi f / f_s)$
$\alpha = s / (2Q)$
$r = 1 / (1 + \alpha)$

$a_0 = 0.5(1 - c)r$
$a_1 = (1 - c)r$
$a_2 = a_0$
$b_1 = -2cr$
$b_2 = (1 - \alpha)r$

The filter outcome was generated using following method:

$$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} - b_1y_{n-1} - b_2y_{n-2}$$

Y is the output signal of filter and X is the input signal.

3.2 Code implementation

Importing necessary libraries

```python
import numpy as np
import matplotlib.pyplot as plt
import IPython.display as ipd
from scipy import signal
```

Defining functions responsible for creating various wave types

```python
# Length is always given in seconds!!!
samplerate = 44100

# Sine wave generator
def sine_wave(freq, length):
    t = np.linspace(0, length, int(length*samplerate), endpoint=False)
    return 0.5*np.sin(2*np.pi*freq*t)

# Rectangular wave generator
def rectangular_wave(freq, length, amplitude):
    if amplitude > 1:
        amplitude = 1
    elif amplitude <= 0.1:
        amplitude = 0.1
    sin = sine_wave(freq, length)
    rect_wave = []
    for i in range(len(sin)):
        if sin[i] > 0:
            y = amplitude
        else:
            y = 0
        rect_wave.append(y)
    return rect_wave
```

```python
# Sawtooth wave generator
def sawtooth_wave(freq, length):
    length = int(44100*length)
    t = np.linspace(0, 1, length)
    return signal.sawtooth(2 * np.pi * freq * t)

# Triangular wave generator
def triangular_wave(freq, length):
    length = int(44100*length)
    t = np.linspace(0, 1, length)
    return signal.sawtooth(2 * np.pi * freq * t, 0.5)

# White noise generator
def white_noise(length, amplitude):
    length = int(length*44100)
    return np.random.uniform(-amplitude, amplitude, length)
```

Creating sound wave that is sum of all previously mentioned waves

```
# Overall result generator function
def wave_fun(freq, length, amplitude):
    sine = sine_wave(freq, length)
    rectangule = rectangular_wave(freq, length, amplitude)
    saw = sawtooth_wave(freq, length)
    triangular = triangular_wave(freq, length)
    noise = white_noise(length, amplitude)
    return sine+rectangule+saw+triangular+noise
```

Creating the sound of demanded frequency, length (in seconds) and amplitude. These can be freely changed by the user.

```
# Complex sound generating
freq = [391.9, 329.6, 329.6, 349.6, 293.7, 293.7, 261.6, 329.6, 391.9]
length = [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2, 0.2, 0.4]
amplitude = [1, 1, 0.5, 0.5, 1, 1, 1, 1, 0.5]
final_sound = []
for i in range(len(length)):
    fin = wave_fun(freq[i], length[i], amplitude[i])
    for j in range(len(fin)):
        final_sound.append(fin[j])
```

Implementing the lowpass filter algorithm

```
# Lowpass filter implementation
result = final_sound
f = 2
fs = 200
Q = 0.5
s = np.sin(2*np.pi*f/fs)
c = np.cos(2*np.pi*f/fs)
a = s / 2*Q
r = 1 / (1+a)

a0 = 0.5*(1-c)*r
a1 = (1-c)*r
a2 = a0
b1 = -2*c*r
b2 = (1-a)*r
filtered_result = []
for i in range(len(result)):
    if i > 2:
        filtered_result.append(a0*result[i] + a1*result[i-1] +
                               a2*result[i-2] - b1*filtered_result[i-1] - b2*filtered_result[i-2])
    else:
        filtered_result.append(a0*result[i])
```

The results were presented in the form of graph using that algorithm and matplotlib libraries:

```
plt.title("Generated and filtered wave sample")
plt.plot(filtered_result[:600], label="Filtered result")
plt.plot(result[:600], label="Generated result")
plt.legend()
plt.show()
```

The sound was generated with use of IPython.display libraries:

```
ipd.Audio(final_sound, rate=samplerate)
```

3.3 Results

As a result of this part, the sample wave has been generated and the melody of that has been presented along with the plot of the data. All implemented solution were working properly. The synthesiser of wave data was correctly generating demanded data with given frequency, length and amplitude. Next the outcome of that process were successfully filtered by lowpass filter.

The outcome of that part of task can be seen at Figure 2. It is seen that filtered data was more smooth, the noise were erased successfully and clear tone was created.

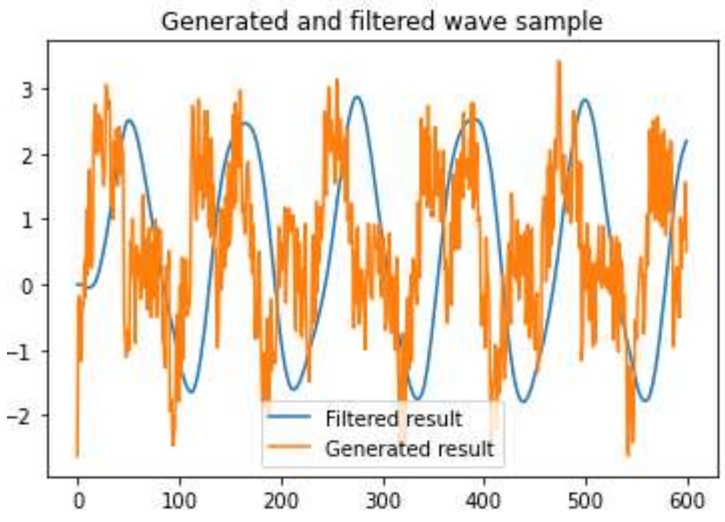


Figure 2 Filtration result

4. Generating of the ocean wave imitation sample

4.1 Theoretical introduction

The ocean wave was created with use of white noise generator data summed up with LFO (Low Frequency Oscillator) data. This sound wave was later passed to the lowpass filter and final result was then obtained. The algorithm used for that implementation looks as follows:

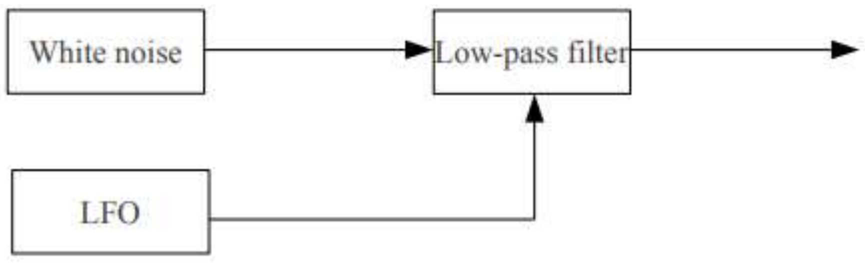


Figure 3 Ocean wave generating algorithm

4.2 Code implementation

First two wave generator were used. Sine wave (LFO wave) and white noise generators.

```
# Sine wave generator
def LFO_wave(freq, length):
    t = np.linspace(0, length, int(length*samplerate), endpoint=False)
    return 400+200*np.sin(1*np.pi*freq*t)
```

```
# White noise generator
def white_noise(length, amplitude):
    length = int(length*44100)
    return np.random.uniform(-amplitude, amplitude, length)
```

After obtaining outcome from both of the generators, it has been later summed up and the ocean wave sample has been created.

```
LFO = LFO_wave(0.5, 4)
noise = white_noise(4, 2)
ocean_wave = noise
```

To obtain the proper result, created ocean wave sample was passed to the lowpass filter algorithm that erased unnecessary noise from the data.

```
result = ocean_wave
ocean_wave_filtered = []
for i in range(len(ocean_wave)):
    f = LFO[i]
    fs = 44100
    Q = 5
    s = np.sin(2*np.pi*f/fs)
    c = np.cos(2*np.pi*f/fs)
    a = s / 2*Q
    r = 1 / (1+a)

    a0 = 0.5*(1-c)*r
    a1 = (1-c)*r
    a2 = a0
    b1 = -2*c*r
    b2 = (1-a)*r
    if i > 2:
        ocean_wave_filtered.append(a0*result[i] + a1*result[i-1] + a2*result[i-2] - b1*ocean_wave_filtered[i-1]
                                   - b2*ocean_wave_filtered[i-2])
    else:
        ocean_wave_filtered.append(a0*result[i])
```

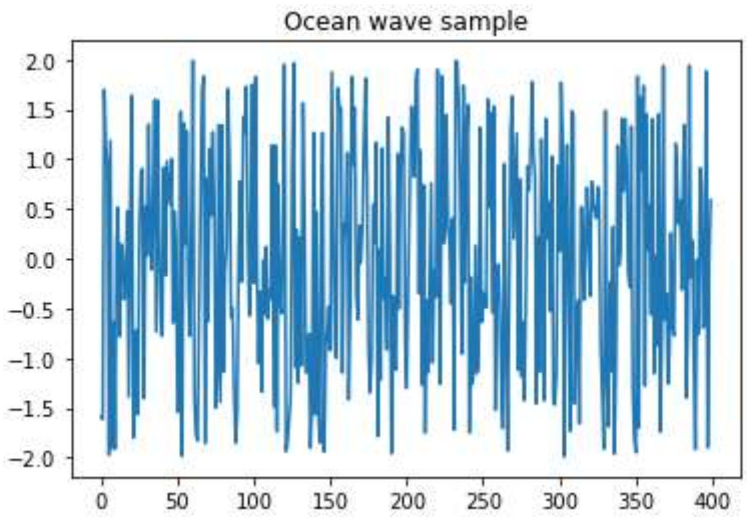
The results was presented as plots containing not filtered and filtered data and their comparison.

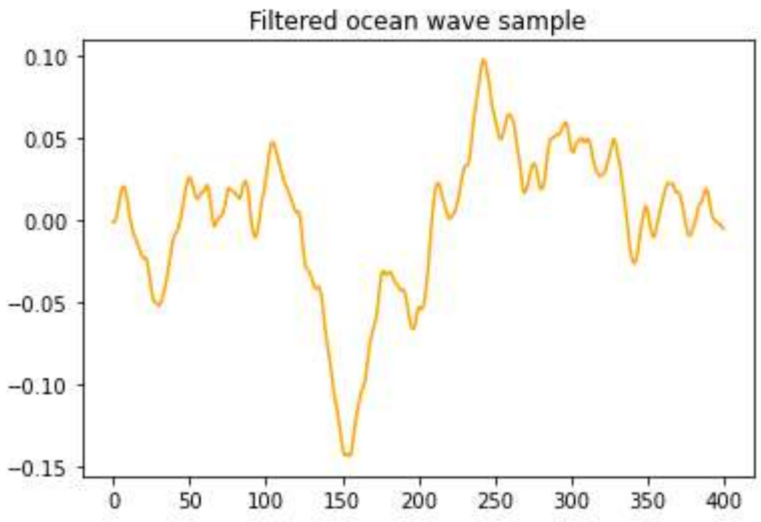
```
plt.title("Ocean wave sample")
plt.plot(ocean_wave[:400])
plt.show()
plt.title("Filtered ocean wave sample")
plt.plot(ocean_wave_filtered[:400], color="orange")
```

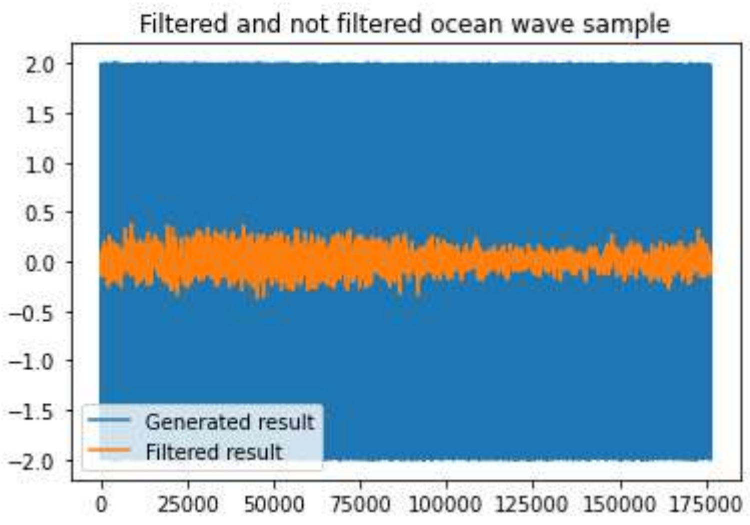
```
plt.title("Ocean wave sample")
plt.plot(ocean_wave, label="Generated result")
plt.plot(ocean_wave_filtered, label="Filtered result")
plt.legend()
plt.show()
```

4.3 Results

As a result, the sound imitating the sound waves has been created. The results of proper working of filtering algorithm can be seen in the plots below. Also the result contain creating the actual sound sample that confirms proper working of an algorithm.






Filtered and not filtered ocean wave sample
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
25000
50000
75000
100000
125000
150000
175000
Generated result
Filtered result