Jan Niclas Ruppenthal
1481198

Heterogenous Computing – Audiodateien

Dieses Dokument stellt die verschiedenen Varianten zur Generierung von Audiodateien in **Generate/WAV_Generator.py** vor. Hierbei wird auch das Tool Audacity[1] verwendet, um das Signal der Dateien zeigen zu können. Im Python-Skript zur Generierung von WAV-Dateien wurden elf verschiedene Varianten implementiert:

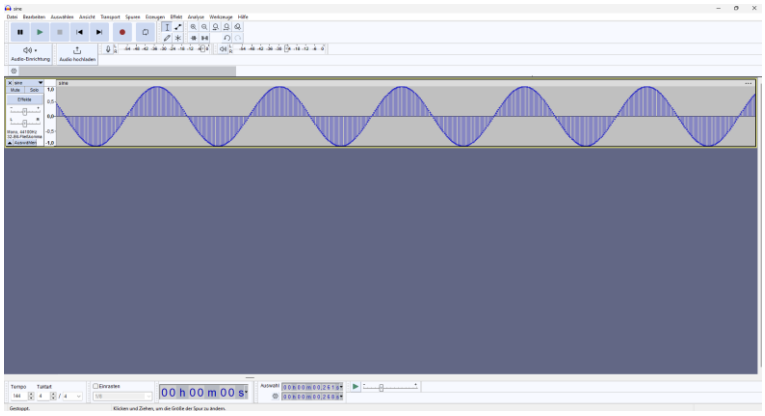
1. Sinuskurve
2. Additive Sinuskurven
3. Amplitudenmodulation
4. Frequenzmodulation
5. Segmenten von Sinuskurven
6. Sinuskurve mit einer Hüllkurve
7. Sinuskurve mit beliebig vielen Harmonischen
8. Dreieckssignal
9. Viereckssignal
10. Zirpe (Signal mit Start- bis Endfrequenz)
11. Noise (zufälliges Signal)

Hierbei hatte ich einfach Spaß mir verschiedene Testszenarien von ChatGPT generieren zu lassen. Außerdem ist es unfassbar interessant, wie man mit wenig Quellcode diverse Audiosignale erzeugen kann.

Sinuskurve

```
def sine_wave(frequency):
    return 0.5 * np.sin(2 * np.pi * frequency * timestamps)
```

In dieser Methode wird eine Sinuskurve mit der Frequenz **frequency** erstellt. Das entstehende Signal mit 440 Hz sieht folgendermaßen aus:



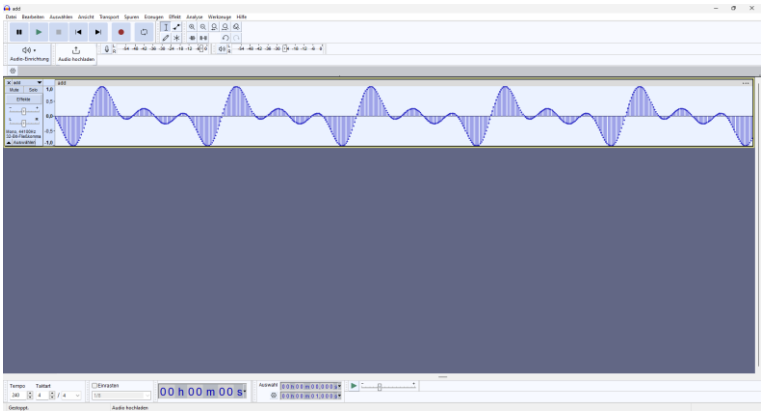
[1] https://www.audacityteam.org/ (Letzter Zugriff am 12.07.2024)

Jan Niclas Ruppenthal
1481198

Additive Sinuskurven

```
def add_sine_wave(frequencies):
    signal = np.zeros_like(timestamps)
    for frequency in frequencies:
        signal += 0.5 * np.sin(2 * np.pi * frequency * timestamps)
    return signal
```

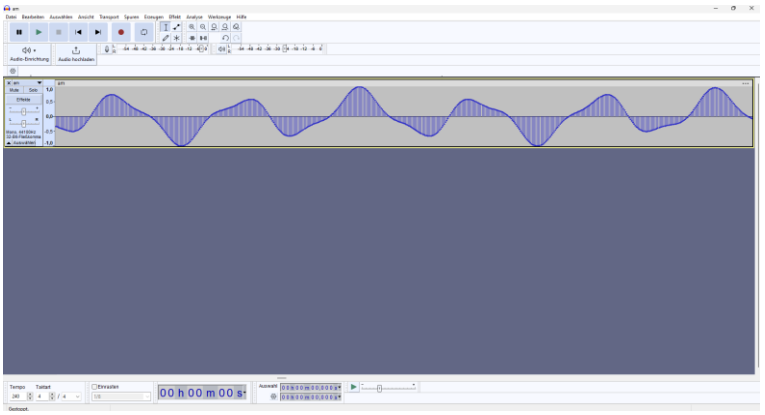
Hierbei werden diverse Sinuskurven mit Frequenzen aus **frequencies** miteinander addiert. Ein Beispiel mit den Frequenzen 400 und 800 und 1200 Hz sieht folgendermaßen aus:



Amplitudenmodulation

```
def am_wave(carrier_freq, mod_freq):
    carrier = np.sin(2 * np.pi * carrier_freq * timestamps)
    modulator = 1 + 0.5 * np.sin(2 * np.pi * mod_freq * timestamps)
    return 0.5 * carrier * modulator
```

In der obigen Methode werden die Daten für eine Audiodatei mithilfe der Amplitudenmodulation erzeugt. Die Trägerfrequenz **carrier_freq** ist die Frequenz des Trägersignals, das moduliert wird. Der zweite Parameter ist die Modulationsfrequenz. Also die Frequenz des Modulatorsignals, welches die Amplitude des Trägersignals variiert. Ein Signal mit einer Trägerfrequenz von 440 Hz und einer Modulationsfrequenz von 600 Hz wird in der folgenden Abbildung dargestellt:

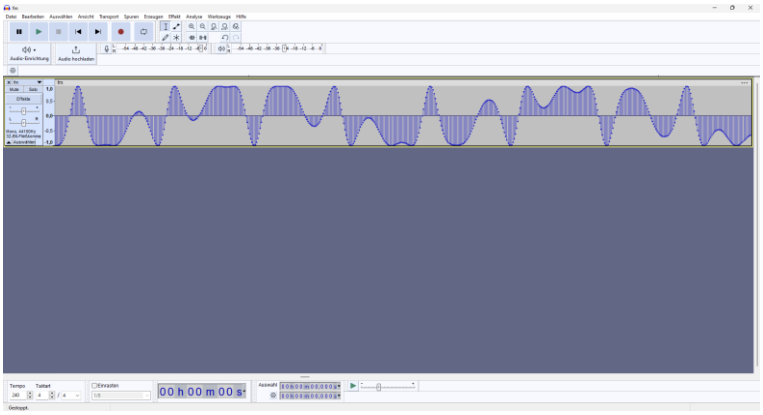


Jan Niclas Ruppenthal
1481198

Frequenzmodulation

```
def fm_wave(carrier_freq, mod_freq, mod_index):
    return 0.5 * np.sin(2 * np.pi * carrier_freq * timestamps + mod_index *
np.sin(2 * np.pi * mod_freq * timestamps))
```

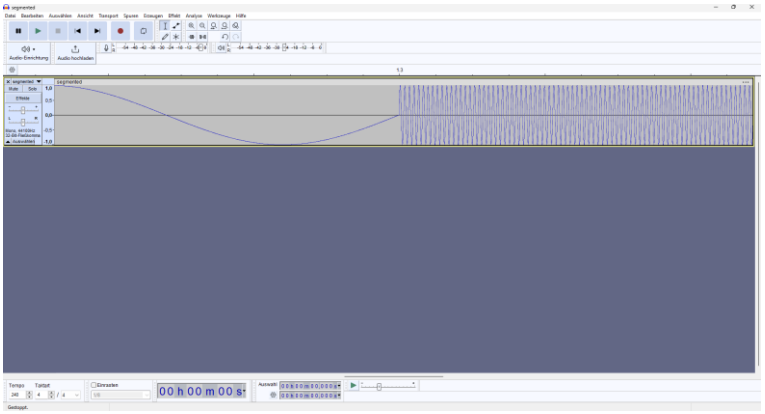
Im Gegensatz zur Amplitudenmodulation wird hier nun die Frequenz des Trägersignals variiert. Die ersten beiden Parameter in der Methode **fm_wave** haben die gleiche Semantik wie die Parameter der vorherigen Methode zur Amplitudenmodulation. Hierbei dient der Index **mod_index** als Faktor für das Ausmaß der Frequenzvariation. Je höher dieser Faktor gewählt wird, desto höher sind die Frequenzabweichungen des Trägersignals. Die nächste Abbildung verdeutlicht ein Signal mit einer Trägerfrequenz von 440 Hz und einer Modulationsfrequenz von 600 Hz und einem Faktor von 2:



Segmente von Sinuskurven

```
def segmented_sine_wave(frequencies, segment_duration):
    signal = np.array([])
    for frequency in frequencies:
        timestamps = np.linspace(0, segment_duration, int(sample_rate * segment_duration),
endpoint=False)
        segment = 0.5 * np.sin(2 * np.pi * frequency * timestamps)
        signal = np.concatenate((signal, segment))
    return signal
```

Mit dieser Methode kann ein Signal generiert werden, welches aus verschiedenen Segmenten besteht. Hierbei werden Sinuskurven von einer Dauer von **segment_duration** erstellt und miteinander verkettet. Der Parameter **frequencies** ist eine Liste von Frequenzen. In der folgenden Abbildung wurde eine Sinuskurve mit 10 Hz mit einem Signal mit 1000 Hz verkettet.

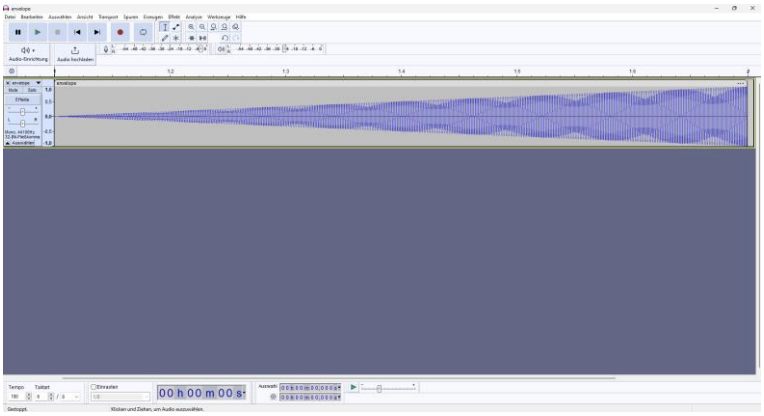


Jan Niclas Ruppenthal
1481198

Sinuskurve mit einer Hüllkurve

```
def sine_wave_with_envelope(frequency):
    envelope = np.linspace(0, 1, int(sample_rate * duration))
    sine_wave = 0.5 * np.sin(2 * np.pi * frequency * timestamps)
    return sine_wave * envelope
```

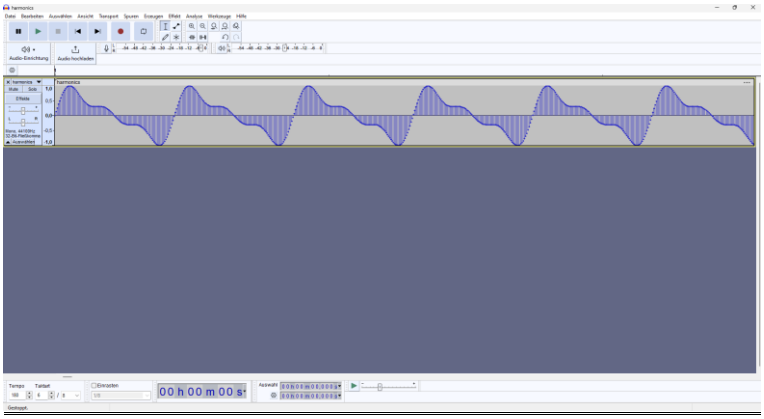
Die Funktion erstellt eine Sinuskurve, deren Amplitude von 0 auf 0.5 ansteigt, die durch die Hüllkurve erreicht wird. Dabei ist die Hüllkurve nur eine lineare Funktion, die gleichmäßig von 0 auf 1 steigt. Für die Darstellung eines Beispiels wurde ein Signal mit einer Frequenz von 440 Hz generiert:



Sinuskurve mit Harmonischen

```
def harmonics_wave(fundamental_freq, num_harmonics):
    signal = np.zeros_like(timestamps)
    for i in range(1, num_harmonics + 1):
        signal += (0.5 / i) * np.sin(2 * np.pi * fundamental_freq * i * timestamps)
    return signal
```

Die obige Funktion generiert eine Sinuskurve mit einer Frequenz **fundamental_freq** und **num_harmonics** Harmonischen. Hierbei werden die notwendigen Frequenzen durch die Anzahl der Harmonischen ermittelt. Jede Sinuskurve wird miteinander addiert. In der folgenden Abbildung wurde ein Signal von 440 Hz mit drei Harmonischen generiert:

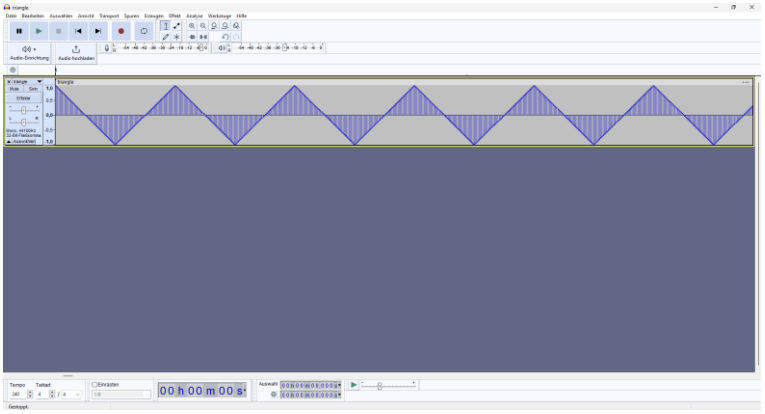


Jan Niclas Ruppenthal
1481198

Dreieckssignal

```
def triangle_wave(frequency):
    period = 1 / frequency
    t_normalized = (timestamps % period) / period
    return 0.5 * (2 * np.abs(2 * t_normalized - 1) - 1)
```

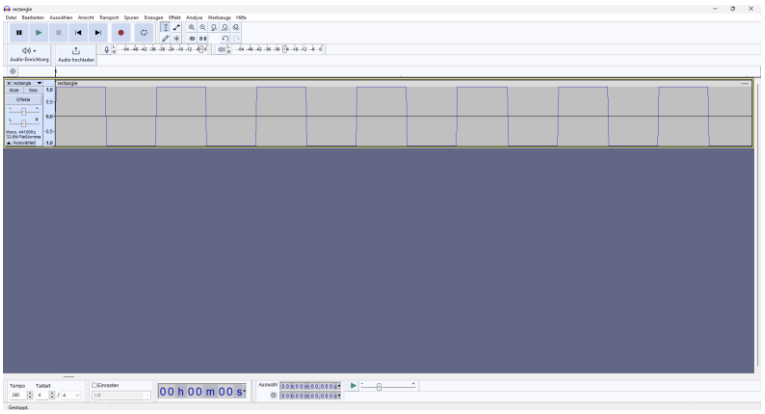
Hier wird ein Dreieckssignal mit der Frequenz **frequency** erzeugt. Das folgende Beispiel enthält eine Frequenz von 440Hz:



Viereckssignal

```
def rectangle_wave(frequency):
    return 0.5 * np.sign(np.sin(2 * np.pi * frequency * timestamps))
```

Durch die Funktion, die das Vorzeichen der übergebenen Zahl ermittelt, wird eine Sinuskurve in ein Viereckssignal umgewandelt. Wie auch beim Dreieckssignal enthält auch das generierte Signal zur Darstellung auch 440 Hz:

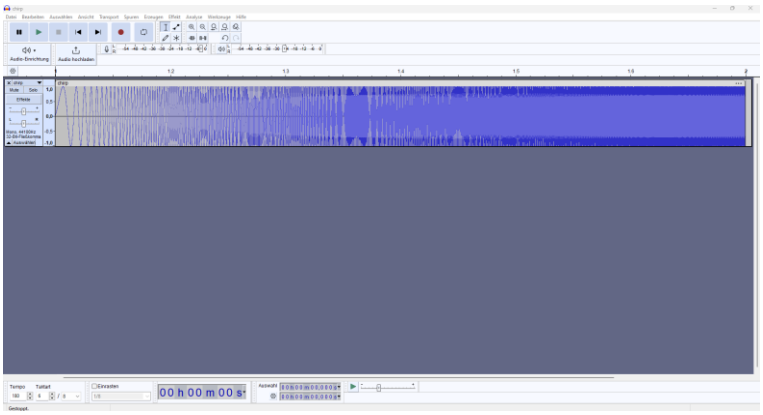


Jan Niclas Ruppenthal
1481198

Chirp

```
def chirp_wave(start_freq, end_freq):
    return 0.5 * np.sin(2 * np.pi * (start_freq + (end_freq - start_freq) * timestamps / duration) * timestamps)
```

In der vorletzten Funktion wird ein Signal zurückgegeben, dessen Frequenz im Laufe der Zeit von **start_freq** bis **end_freq** ansteigt oder herabfällt. Zur Darstellung eines Beispiels wurde ein Signal generiert, welches von 400 bis 1200 Hz ansteigt:



Zufälliges Signal

```
def noise():
    return 0.5 * np.random.normal(0, 1, int(sample_rate * duration))
```

In der letzten Funktion wird ein zufälliges Signal durch eine Methode zur Randomisierung erzeugt. Ein Beispiel eines zufälligen Signals sieht folgendermaßen aus: