University of South-Eastern Norway

Spectra, spectrograms, and aliasing

TSE2280 Signal Processing

Lab 2, spring 2025

1 Introduction

1.1 Background and Motivation

This lab is based on two labs that that accompany the course text-book *DSP First* by McClellan et al. [1], *Lab P-4: AM and FM Sinusoidal Signals* and *Lab S-8: Spectrograms: Harmonic Lines & Chirp Aliasing*[2]. The lab demonstrates concepts from Chapters 3 and 4 in the text-book, covering spectra, spectrograms, and aliasing. The lab will also give practical experience with some common signals, such as square and triangle waves, beats, and chirps. The original labs have been modified and converted from Matlab to Python.

Listening to the sound of a signal can be informative. The sound of the signals can be played using the sound card in the computer and compared to what we see in the spectra and spectrograms.

1.2 Software Tools: Python with Spyder and JupyterLab

Python is used for for programming, with Spyder [3] as the programming environment, and JupyterLab for reporting. The signals are represented as NumPy arrays and plotted in Matplotlib. This lab also introduces spectral analysis tools from the signal processing modules in SciPy, `scipy.fft` and `scipy.signal`, and the module `sounddevice` to play the sounds over the sound card.

The recommended way to import the Python modules is shown in Table 1. You are free to do this in other ways, but the code examples in this text assumes modules are imported as described here.

2 Theory with Programming Examples

2.1 Sinusoids

Lab 1 introduced sinosoidial waveforms represented as

$$x(t) = A\cos(\omega t + \phi) = \mathrm{Re}\left\{Ae^{j(\omega t+\phi)}\right\} = \mathrm{Re}\left\{Xe^{j\omega t}\right\} \qquad , \qquad X = Ae^{j\phi} \tag{1}$$

where A is the amplitude, $\omega = 2\pi f$ is the angular frequency, f is the frequency, and ϕ is the phase. X is a *phasor* or *complex amplitude* that includes both the amplitude and the phase of the signal. In Lab 1, you also wrote a function to create a signal $x_s(t)$as a sum of several sinusoids described by frequencies f_k and complex amplitudes X_k

$$x_s(t) = \sum_{k=1}^{N} A_k \cos(2\pi f_k t + \phi_k) = \mathrm{Re}\left\{\sum_{k=1}^{N} X_k e^{j2\pi f_k t}\right\} \quad , \qquad X_k = A_k e^{j\phi_k} \ . \tag{2}$$

2.2 Beat and Amplitude Modulated Signals

A beat signal is a sum of two sinusoids with frequencies f_1 and f_2 that are close. Chapter 3 in the textbook [1] shows how this can be interpreted as a sinusoid with a high frequency $f_c = \frac{1}{2}(f_2 + f_1)$

Table 1. Recommended format for importing the Python modules. NumPy is used to manipulate signals as arrays, Matplotlib to plot results, and SciPy for signal processing.

```
import numpy as np                    # Handle signals as arrays
import matplotlib.pyplot as plt       # Show results as graphs and images
from math import pi, cos, sin, tan    # Mathematical functions on scalars
from cmath import exp, sqrt           # Complex mathematical functions on scalars

from scipy.fft import fft, fftshift, fftfreq    # FFT and helper functions
from scipy import signal              # Signal processing functions

import sounddevice as sd              # Play NumPy array as sound
```

enclosed in a slowly varying envelope with frequency $f_\Delta = \frac{1}{2}(f_2 - f_1)$,

$$x_b(t) = \cos(2\pi f_1 t) + \cos(2\pi f_2 t) = 2\cos(2\pi f_\Delta t)\cos(2\pi f_c t)\,. \tag{3}$$

Amplitude modulated (AM) signals used in radio trasnmission can be viewed as an extension to the beat signal. The AM-signal is written as

$$x_{AM}(t) = (1 + M\cos(2\pi f_0 t))\cos(2\pi f_c t) = \cos(2\pi f_0 t) + \tfrac{1}{2}\cos(2\pi(f_c - f_0)t) + \tfrac{1}{2}\cos(2\pi(f_c + f_0)t)\,. \tag{4}$$

2.3 Frequency Modulated Signals, Chirps

In a sinusoid with constant frequency, the argument of the cosine-function $\Psi = 2\pi f t + \phi$ varies linearly with time. The time derivative of Ψ is its angular frequency $\omega = \frac{\mathrm{d}\Psi}{\mathrm{d}t} = 2\pi f$. This can be generalised by setting the argument to the cosine-function to an arbitrary function $\Psi(t)$,

$$x_{FM}(t) = A\cos(\Psi(t))\,, \tag{5}$$

and defining the instantaneous frequency as

$$\omega_i = \frac{\mathrm{d}\Psi(t)}{\mathrm{d}t}\,, \qquad\qquad f_i = \frac{\omega_i}{2\pi} = \frac{1}{2\pi}\frac{\mathrm{d}\Psi(t)}{\mathrm{d}t}\,. \tag{6}$$

A *linear chirp*, often just called a *chirp*, is a sinusoid where the frequency changes linearly with time. The instantaneous phase $\Psi(t)$ for a linear chirp is a quadratic function of time,

$$\Psi(t) = 2\pi\mu t^2 + 2\pi f_0 t + \phi\,, \tag{7}$$

and the instantaneous frequency is

$$f_i = 2\mu t + f_0\,. \tag{8}$$

This is an example of a frequency modulated (FM) signal. The linear variation of the frequency can produce an audible sound similar to a siren or a chirp, giving this class of its name.

An example of Python code to generatte a chirp is given in Table 2.

2.4 Spectra and spectrograms

2.4.1 Spectrum and Fourier Coefficients

The spectrum of a signal is a representation of the frequencies present in the signal and can be calculated by finding the Fourier coefficients a_k. For a sampled signal $x[n]$, this can be done efficiently by using the FFT-algorithm, which is available as the function `fft` in the SciPy module also called `fft`. Some remarks about how the results returned from the FFT-algorithm are related to the Fourier coefficients a_k are:

1. FFT returns coefficients as complex numbers, with magnitude and phase .
2. The result returned from FFT are the Fourier coefficients a_k multiplied by the number of samples n_s. The coefficients a_k are found by dividing the FFT output by the number of samples.

Table 2. Example code to generate a chirp.

```
fs = 11025          # Sample rate, 1/4 of the standard 44.1 kHz
ts = 1/fs           # Sample interval
duration = 1.8      # Signal duration
t = np.arange(0, duration, ts)  # Time vector
f0 = 100            # Start frequency
mu = 500            # Variation mu
phi = 100           # Initial phase

psi = 2*pi*mu*t**2 + 2*pi*f0*t + phi    # Instantaneous phase

A = 7.7             # Amplitude
x = np.real(A*np.exp(1j*psi*t))         # Signal
```

Table 3. Example code to calculate and plot the Fourier coefficients of an arbitrary signal $x(t)$ using the `fft` module in SciPy. The helper function `fftfreq` is used to find the frequency vector corresponding to the Fourier coefficients, while `fftshift` puts the positive and negative coefficients in the corrrect order. The definition of the FFT algorithm requires the output to be scaled with the number of samples to get the correct magnitudes. The last part plots the Fourier coefficients in two graphs, one for magnitude and one for phase. Note that the phase can take random values where the magnitude is very low, as this is mostly noise.

```
# Calculate FFT and frequency vector and order results
n_samples = len(x)             # No. of samples in signal
ft_x = fft(x)/n_samples        # Fourier coefficients, correctly scaled
f = fftfreq(n_samples, 1/fs)   # Frequency vector
f = fftshift(f)                # Move negative frequencies to start
ft_x = fftshift(ft_x)

# Plot Fourier coefficients
fig = plt.figure(figsize=([16, 6]))     # Define figure for plots

ax1 = fig.add_subplot(1, 2, 1)  # Subplot for magnitudes
ax1.stem(f, np.abs(ft_x))       # Magnitude of spectral components as stem-plot
ax1.set(xlabel="Frequency␣[Hz]",
        ylabel="Magnitude")
ax1.grid(True)

ax2 = fig.add_subplot(1, 2, 2)  # Subplot for phase
ax2.stem(f, np.angle(ft_x))     # Phase of spectral components as stem-plot
ax2.set(xlabel="Frequency␣[Hz]",
        ylabel="Phase␣[radians]")
ax2.grid(True)
```

3. FFT returns both positive and negative frequency components. The number of frequency values returned from FFT is twice the number of samples in the input vector.
4. The Fourier coefficients returned from FFT are arranged with the negative coefficients *after* the positive coefficients. The helper function `fftreshape` reorganize the results so the negative frequencies come first.
5. The frequencies corresponding to the Fourier coefficients are returned from the function `fftfreq`.
6. The FFT algorithm works by breaking down the number of points in prime factors, it is most efficient if the number of points contain only one prime factor. This prime factor is often 2, and FFTs are often evaluated for sequences of 2^n points, e.g., 256, 512, 1024, 2048, etc... The mathematical details can be found in the text-book [1].

Example code for how to calculate the Fourier coefficients, order them correctly, and plot the magnitude and phase is shown in Table 3.

Table 4. Example code to calculate and plot the power spectral density, PSD, of a signal $x(t)$ using the `periodogram` function in SciPy, `scipy.signal.periodogram`. The code takes a a signal `x` and its time vector `t`, calculates the PSD, and plots the signal in the upper graph and its power spectrum in the lower graph.

```
# Calculate PSD
f, pxx = signal.periodogram(x, fs)   # Calculate PSD (pxx) and frequencies (f)

# Plot result
fig = plt.figure(figsize = [16, 8])  # Define figure for plots

ax1 = fig.add_subplot(2, 1, 1)  # Subplot for signal
ax1.plot(t, x)                  # Signal x as function of time t
ax1.set(xlabel="Time [s]",
        ylabel="Amplitude")
ax1.grid(True)

ax2 = fig.add_subplot(2, 1, 2)  # Subplot for power spectral density
ax2.plot(f, pxx)                # Power spectral density
ax2.set(xlabel="Frequency [Hz]",
        ylabel="PSD [V^2/Hz]")
ax2.grid(True)
```

2.4.2 Power Spectrum

The power associated with frequency component k is $\frac{1}{2}|a_k|^2$. This can be expressed as a *power spectral density, PSD*, or power per frequency interval, by normalising with the frequency interval $\Delta f = 1/T_0$. The unit of the PSD is *amplitude square per frequency*, e.g., if $x(t)$ is measured in Volt, PSD has unit V^2/Hz. This scaling makes the PSD independent of T_0 and sample rate f_s, making comparison between spectra easier.

For a real-valued signal, the positive and negative frequency coefficients are complex conjugates, $a_{-k} = a_k^*$ and $|a_{-k}|^2 = |a_k|^2$. In this case, the power in the negative and positive frequencies can be added to a *single-sided PSD*. This is the most common way of presenting power spectra, the power spectral density P_{xx} is expressed by the Fourier coefficients as

$$P_{xx} = \begin{cases} a_0^2 & k = 0 \\ \frac{1}{2}|a_k|^2 & k > 0 \end{cases} \quad . \tag{9}$$

This theory is built into a SciPy function `scipy.signal.periodogram`. All all we need to do to find the correctly scaled power spectral density of a signal is to call this function, which also returns the frequency vector. Example code for this is given in Table 4. The calculation of the PSD is only one line, the rest is for plotting the results.

2.4.3 Spectrogram

A spectrum that changes with time can be illustrated with a spectrogram, see Chapter 3-6 in the course text-book [1]. The spectrogram is found dividing the time signal in short intervals of length T_0, finding the PSD over each segment, and plotting the result on atwo-dimensional intensity plot with time and frequency on the axes. Some comments to the use of spectrograms are

1. Python offers several versions of spectrogram. One version that is straightforward and simple to use is found in SciPy, `scipy.signal.spectrogram`.
2. Spectrograms are numerically calculated over short segments of the signal and give an estimate of the time-varying frequency content of the signal. The finite length of the time segments limits the frequency resolution in the spectrogram, see Section 2.5.
3. Spectrograms can be difficult to configure. Critical parameters are the length of segments and the scales of the frequency axis and intensity display. Different settings can create spectrograms that look different, although presenting the same data.

Example of code for plotting a spectrogram is shown in Table 5.

Table 5. Example code for plotting the spectrogram of a signal `x` sampled at sample rate `t`. A good looking spectrogram requires its parameteres to be correctly configured. The most critical parameters are the segment length `n_segment`, the maximum on the frequency axis `f_max`, and the dynamic range on the intensity plot, defined by the minimum value `s_min`. Note the parameter `detrend=False`. Without this, the data will be 'detrended' by removing DC-components.

```
# Configure spectrogram
n_segment = 1024  # Length of segment
f_max = 400       # Maximum frequency to show in spectrogram
s_min = -40       # Minimum on the intensity plot. Lower values are 'black'

# Calculate spectrogram
f, t, sx = signal.spectrogram(x, fs, nperseg=n_segment, detrend=False)
sx_db = 10*np.log10(sx/sx.max())   # Convert to dB

# Plot spectrogram
fig = plt.figure(figsize=(16, 8))  # Define figure for results
ax = fig.add_subplot(1, 1, 1)

sx_image = ax2.pcolormesh(t, f, sx_db, vmin=s_min)

ax.set(xlabel = "Time [s]",
       ylabel = "Frequency [Hz]",
       ylim = (0, f_max) )

fig.colorbar(sx_image, label="Magnitude [dB]")  # Colorbar for intensity scale
```

2.5 Resolution in Time and Frequency

A signal $x(t)$ over an interval T_0 can be written as a sum of complex exponentials where the frequencies are the harmonics of the fundamental $f_k = kf_0 = k/T_0, k = 0, 1, 2, \ldots$,

$$x(t) = \sum_{k=-\infty}^{+\infty} a_k e^{j2\pi kt/T_0} \,. \tag{10}$$

The strength of the frequency components are given by the Fourier coefficients a_k. They can be found from the signal $x(t)$ by the following equation from the text-book [1]

$$a_k = \frac{1}{T_0} \int_0^{T_0} x(t) e^{-j2\pi kt/T_0} \mathrm{d}t \,. \tag{11}$$

This gives an important relation between resolution in time and frequency. The frequencies of the Fourier coefficients are $0, f_0, 2f_0, 3f_0, \ldots$, so the spacing between them is $\Delta f = f_0 = 1/T_0$. A good resolution in frequency, a low Δf, requires a long observation time, a large T_0. The segment length T_0 is the resolution in time, $\Delta t = T_0$. This gives the important relation between resolution in time and frequency,

$$\Delta f = 1/T_0 \qquad \Delta t = T_0 \qquad \Delta f = \frac{1}{\Delta t} \,. \tag{12}$$

This shows that we must compromise between resolution in time Δt and resolution in frequency Δf, and how this is controlled by the segment length T_0. The balance between Δf and Δt is especially important when presenting data as spectrograms, as you will test out in the lab exercises.

2.6 The Decibel-Scale

A logarithmic scale allows visualization of a wider dynamic range than a linear scale and is preferred when the data set spans from very large to very small values. The decibel (dB) scale is the standard logarithmic scale in engineering and is defined as

$$L_{dB} = 10 \log_{10} \left(\frac{W}{W_{ref}} \right) = 20 \log_{10} \left(\frac{v}{v_{ref}} \right) \tag{13}$$

where v is an amplitude value (voltage, current, etc.) and W is power or energy, so that $W/W_{ref} = (v/v_{ref})^2$. A value in dB is always defined relative to a reference value, v_{ref} or W_{ref}. 0 dB is this reference value. This reference can be a predefined value (e.g., 1 V or 1 mW), or an input or maximum value.

Spectral data are mostly presented in dB. Since we often only interested in the relative variation between the spectral components, the reference value 0 dB is often chosen as the maximum value in the data set. This results in all values in the spectrum or spectrogram being negative decibels, where 0 dB is the maximum. The scale minimum gives the dynamic range, the span between the largest and smallest value presented. Typical values for this are -40 dB or -60 dB.

Decibels are never used for high-precision values. Some important dB values are listed in Table 6.

Table 6. List of important dB-values.

	Amplitude ratio	Power ratio
dB	v/v_{ref}	W/W_{ref}
0	1	1
-3	$\frac{1}{\sqrt{2}} = 0.71$	$\frac{1}{2} = 0.5$
-6	$\frac{1}{2} = 0.5$	$\frac{1}{4} = 0.25$
-10	$\frac{1}{\sqrt{10}} = 0.32 \approx \frac{1}{3}$	$\frac{1}{10} = 0.1$
-20	$\frac{1}{10} = 10^{-1}$	$\frac{1}{100} = 10^{-2}$
-40	$\frac{1}{100} = 10^{-2}$	$\frac{1}{10000} = 10^{-4}$
-60	$\frac{1}{1000} = 10^{-3}$	$\frac{1}{1000000} = 10^{-6}$

2.7 Fourier Series of Square and Triangle Waves

Two important signals are the *square* and *triangular* waves shown in Figure 1. The Fourier coefficients for these can be calculated from (11), the results are

$$\text{Square wave } x_s(t) \qquad a_k = \begin{cases} \frac{2}{j\pi k} & k = \pm 1, \pm 3, \pm 5, \ldots \\ 0 & k = 0, \pm 2, \pm 4, \pm 6, \ldots \end{cases} \tag{14a}$$

$$\text{Triangle wave } x_t(t) \qquad a_k = \begin{cases} \frac{4}{j\pi^2 k^2}(-1)^{(k-1)/2} & k = \pm 1, \pm 3, \pm 5, \ldots \\ 0 & k = 0, \pm 2, \pm 4, \pm 6, \ldots \end{cases} \tag{14b}$$

The even components ($k = \pm 2, \pm 4, \ldots$) vanish for both these wave. This comes out of equation (11), but can also be seen from symmetry.

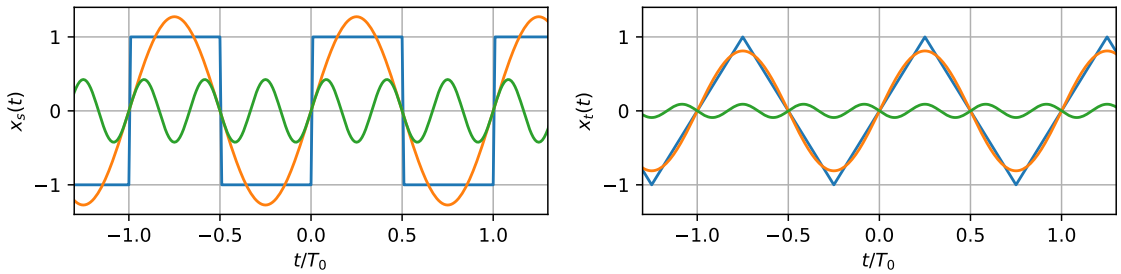

Figure 1. Square and triangle waves with the waves from the 1st and 3rd Fourier coefficients added. The triangle wave is better reproduced by a few Fourier coefficients than the square wave, and its Fourier coefficients decrease more rapidly as k increases.

3 Lab Exercises

Reporting

Collect answers and code in a JupyterLab notebook. Export this to pdf and upload it to Canvas.

You may prefer to do some of the coding in another development tool, such as Spyder. This is actually recommend, as the testing and debugging options are better in Spyder than in JupyterLab. You can load the .py-files from from Spyder into JupyterLab and export them as a separate pdf-file.

3.1 Spectrum and Spectrogram of Sinusoids

This exercise will find the spectra and spectrograms for the two signals generated in Lab 1, use the functions you wrote in this lab. The frequency contents of these two signals does not change with time, so the spectrogram only should contain horizontal lines. The two signals are

a) Single-frequency signal with amplitude A, frequency f, phase ϕ and duration given as

$$A = 10^4 \qquad f = 1.5\,\text{MHz} \qquad \phi = -45^\circ \qquad \text{Duration } 4 \times 10^{-6}\,\text{s}$$

Set the sample rate to exactly 32 samples per period.

b) A signal which is the sum of three sinusoids described by

	k	1	2	3
Frequency	f_k [Hz]	0	100	250
Complex amplitude	X_k	10	$14e^{-j\pi/3}$	$8j$

Set the sample rate to $10\,000\,\text{Samples/s}$ and duration to $0.1\,\text{s}$.

Exercises

1) Plot the two signals and their Fourier coefficients a_k. Use the code in Table 3 as template.

2) Plot the spectrogram of the two signals, using the code in Table 5 as template. Select a segment length and maximum frequency that fits to the signals, you may try different vales.

3.2 Beat

1) Write a Python function to generate a beat signal defined by

$$x(t) = A_1 \cos(2\pi(f_c - f_\Delta)t) + A_2 \cos(2\pi(f_c + f_\Delta)t)\,. \tag{15}$$

Specify the signal by the two amplitudes A_1 and A_2, the centre frequency f_c and the difference frequency f_Δ, the sample rate and the signal duration.

A template for the function header is given in Table 7. You can make this simple by calling the function `make_summed_cos` from Lab 1.

2) Test the function for the input values

$$A_1 = 10 \quad , \quad A_2 = 10 \quad , \quad f_c = 400\,\text{Hz} \quad , \quad f_\Delta = 10\,\text{Hz} \quad , \quad f_s = 11\,025\,\text{Hz} \quad .$$

Set the duration of the signal to $2.0\,\text{s}$, but plot only the first $0.5\,\text{s}$.

Plot the signal and its power spectrum in two subplots. Scale the frequency axis so that the frequencies in the beat are clearly identified.

Comment the result.

3) Play the beat signal as a sound by using the module `sounddevice`. Listen to the signal and comment how it sounds.

4) Change the difference frequency to $f_\Delta = 5\,\text{Hz}$ and $2\,\text{Hz}$. Plot these signals and listen to the sound of them.

Comment how this changed the signal in the time and frequency domains.

Table 7. Template for code to generate a beat from two cosine waves.

```
def beat(A, fc, df, fs, duration)
    """Synthesize a beat tone from two cosine waves.

    Parameters
    ----------
    A: List of floats
        Amplitudes of the two cosine waves
    fc: float
        Centre frequency [Hz]
    df: float
        Difference frequency [Hz]
    fs: float
        Sample rate [samples/s]
    duration
        Duration of signal [s]

    Returns
    -------
    x: 1D NumPy array of float
        Signal as the sum of the frequency components
    t: 1D NumPy array of float
        Time vector [seconds]
    """

    --- Your code comes here ---

    return x, t
```

5) Use your function to generate a new beat signal with the following values

$$A_1 = A_2 = 10 \quad , \quad f_c = 1000\,\text{Hz} \quad , \quad \Delta f = 32\,\text{Hz} \quad , \quad f_s = 11\,025\,\text{Hz} \quad .$$

Set the duration of the signal to $0.26\,\text{s}$ and plot the signal and its spectrum.

6) Plot the spectrogram of this signal using the template from Table 5. Set the segment length to 1024 samples and limit the frequency scale so that the spectral lines are clearly seen.

Comment the result.

7) Do the same for segment lengths 512, 256, and 128 samples.

Comment how the different segment lengths change the appearance of the spectrogram. Relate this to the resolution in time and frequency from Section 2.5.

What is the minimum segment length that can separate the two frequency lines?

3.3 Chirp

This exercise repeats some of the tasks from the previous exercise with a chirp instead of a beat.

1. Write a Python function to generate a chirp signal defined by

$$x(t) = \cos \Psi(t) \qquad \Psi(t) = 2\pi\mu t^2 + 2\pi f_0 t + \phi\,. \tag{16}$$

Specify the signal by start and end frequencies, f_1 and f_2, phase ϕ, sample rate, and duration.

A template for the header of the function is given in Table 8.

2. Test the function for the input values

$$f_1 = 5000\,\text{Hz} \quad , \quad f_2 = 300\,\text{Hz} \quad , \quad f_s = 11\,025\,\text{Hz} \quad .$$

Set the duration of the signal to $3.0\,\text{s}$.

Generate the chirp signal and play it using the module `sounddevice`. Comment how the signal sounds compared to the specification of the chirp.

Table 8. Template for code to generate a chirp from the start and end frequencies.

```
def make_chirp(f1, f2, fs, duration)
    """Synthesize a beat tone from two cosine waves.

    Parameters
    ----------
    f1: float
        Start frequency [Hz]
    f2: float
        End frequency [Hz]
    phase: float
        Constant phase [radians]
    fs: float
        Sample rate [samples/s]
    duration
        Duration of signal [s]

    Returns
    -------
    x: 1D NumPy array of float
        Signal as the sum of the frequency components
    t: 1D NumPy array of float
        Time vector [seconds]
    mu: float
        Frequency slope of chirp, mu
    """

    --- Your code comes here ---

    return x, t, mu
```

3. Plot the spectrogram of this signal using the template from Table 5. Set the segment length to 2048 samples. Comment the result. Is it as expected?

4. Do the same as above for segment lengths 1024, 512, 256, 128, 64, and 4096 samples.

   Comment how the different segment lengths change the appearance of the spectrogram.

   What seems to be the best segment length to resolve this chirp in time and frequency?

5. Generate a new chirp with duration 4 s starting at f_1=100 Hz and ending at f_2=4000 Hz. Set the sample rate to 5000 Hz.

   Play the sound of this chirp using `sounddevice` and plot the spectrogram. Use segment length 512 points.

   Comment the result.

   Change the sample rate to 10 000 Hz and repeat the tasks above. Since the sample rate is doubled, the segment length should also be doubled to get segments with the same duration in time.

   Comment the result. Why do the spectrograms look different?

6. Generate a new chirp with duration 3 s starting at f_1=3000 Hz and ending at the negative frequency f_2=−2000 Hz.

   Listen to the signal. How does the frequency of the sound change?

   Plot the spectrogram of this chirp and explain the result.

   It may be easier to interpret this result if it is displayed as a *two-sided* spectrogram that shows both positive and negative frequencies. This is done by setting the parameter `return_onesided` to `False` and then arrange the negative and positive frequencies correctly using `fftshift`. The code for this is shown below.

```
f, t, sx = spectrogram(x, fs, nperseg=n_segment, return_onesided=False)
f = fftshift(f)
sx = fftshift(sx, axes=0)
```

3.4 Spectra of Square and Triangle Waves

1) Generate a square wave and a triangle wave, both with frequency $100\,\text{Hz}$. Each wave shall have exactly 2 periods. Use at least 100 samples per period to plot the signals.

   Create the signals by using square and triangle wave functions in SciPy, `scipy.signal.square` and `scipy.signal.sawtooth`. Look in the documentation for SciPy to see how to configure them.

2) Calculate and plot the Fourier coefficients of the two waves, use the template code in Table 3.

   Compare the result with the values in (14a) and (14b).

3) Calculate and plot the power spectrum of the two waves, use the template code in Table 4. Plot the spectrum on a decibel-scale.

   Compare this result with the previous result.

4) Calculate and plot the spectrograms of the two waves, use the template code in Table 5. Plot the intensity on a decibel-scale.

   Compare this result with the previous results and comment.

5) Generate a chirp from a square wave. This is done by replacing the cosine-function in the chirp with the square-wave function.

   Let the frequency of the chirp start at $100\,\text{Hz}$ and end at $4000\,\text{Hz}$. Set the sample rate to $11\,025\,\text{Hz}$, and plot the spectrogram on a dB-scale. A two-sided spectrogram may make the interpretation easier.

   Play the sound of the chirp.

   Comment the result.

References

[1] J. H. McClellan, R. Schafer, and M. Yoder, *DSP First*. United Kingdom: Pearson Education Limited, 2nd ed., 2016.

[2] J. H. McClellan, R. Schafer, and M. Yoder, "Lab P-3: Introduction to Complex Exponentials - Direction Finding," tech. rep., 2016.

[3] P. Raybaut, "Spyder," 2024.