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 demonstrates concepts from Chapters 3 and 4 in the course textbook McClellan et al., *DSP First* [1], 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. This lab is based on two labs that that accompany the textbook, *Lab P-4: AM and FM Sinusoidal Signals* [2] and *Lab S-8: Spectrograms: Harmonic Lines & Chirp Aliasing* [3]. The original labs have been modified and converted from Matlab to Python.

1.2 Software Tools: Python with Spyder and JupyterLab

Python is used for for programming, with Spyder [4] as the programming environment and JupyterLab [5] for reporting. The signals are represented as NumPy [6, 7] arrays and plotted in Matplotlib [8, 9]. This lab introduces spectral analysis tools from SciPy [10, 11], using the FFT module `scipy.fft` and the signal processing module `scipy.signal`.

Listening to the sound of a signal can be informative. Sounds can be played using the sound card in the computer and compared to what we see in spectra and spectrograms. The module `sounddevice` is used to play NumPy arrays as sounds.

All programming examples assume the Python modules are imported as described in Table 1.

2 Theory with Programming Examples

2.1 Sinusoids as Complex Exponentials

Lab 1 introduced sinosoidial waveforms represented as

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

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. The sinusoid can also be expressed as the sum of two phasors rotating in opposite directions,

$$x(t) = A\cos(\omega t + \phi) = \tfrac{1}{2}Ae^{j(\omega t+\phi)} + \tfrac{1}{2}Ae^{-j(\omega t+\phi)} = \tfrac{1}{2}Xe^{j\omega t} + \tfrac{1}{2}X^*e^{-j\omega t} \quad , \tag{2}$$

where X^* denotes the complex conjugate. This is the *two-sided spectrum* representation, $\cos\omega t$ contains the positive frequency ω and the negative frequency $-\omega$, each having amplitude $\frac{1}{2}A$. This description corresponds to the output from the FFT-algorithm.

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
```

In Lab 1, you also wrote a function to create a signal $x_s(t)$ as a sum of several sinusoids described by frequencies $\omega_k = 2\pi f_k$ and complex amplitudes X_k

$$\begin{aligned} x_s(t) &= \sum_{k=1}^{N} A_k \cos(\omega_k t + \phi_k) = \mathrm{Re}\left\{ \sum_{k=1}^{N} X_k e^{j\omega_k t} \right\} \\ &= \sum_{k=1}^{N} \left(\tfrac{1}{2} X_k e^{j\omega_k t} + \tfrac{1}{2} X_k^* e^{-j\omega_k t} \right) , \qquad X_k = A_k e^{j\phi_k} . \end{aligned} \tag{3}$$

2.2 Beat and Amplitude Modulated Signals

A beat is a sum of two sinusoids with different but close frequencies f_1 and f_2. Chapter 3 in the textbook [1] shows how this can be interpreted as a sinusoid with a high frequency $f_c = \frac{1}{2}(f_2 + f_1)$ enclosed in a slowly varying envelope with frequency $f_\Delta = \frac{1}{2}(f_2 - f_1)$,

$$x_b(t) = \cos(2\pi f_1 t) + \cos(2\pi f_2 t) = 2\cos(2\pi f_\Delta t)\cos(2\pi f_c t) . \tag{4}$$

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

$$\begin{aligned} x_{AM}(t) &= (1 + M\cos(2\pi f_0 t))\cos(2\pi f_c t) \\ &= \cos(2\pi f_0 t) + \tfrac{1}{2} M \cos(2\pi (f_c - f_0)t) + \tfrac{1}{2} M \cos(2\pi (f_c + f_0)t) , \end{aligned} \tag{5}$$

where M is the degree of modulation, a number between 0 and 1.

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 $\Psi(t)$,

$$x_{FM}(t) = A\cos(\Psi(t)) , \tag{6}$$

and defining the instantaneous frequency as

$$\omega_i = \frac{\mathrm{d}\Psi(t)}{\mathrm{d}t} , \qquad f_i = \frac{\omega_i}{2\pi} = \frac{1}{2\pi}\frac{\mathrm{d}\Psi(t)}{\mathrm{d}t} . \tag{7}$$

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,

$$\Psi(t) = 2\pi\mu t^2 + 2\pi f_0 t + \phi , \tag{8}$$

and the instantaneous frequency is

$$f_i = 2\mu t + f_0 . \tag{9}$$

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 signals of its name. The instantaneous frequency f_i can carry information by letting it vary in more complex patterns, as is done in FM radio transmission.

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

Table 2. Example code to generate a chirp. See definition in (8).

```
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
```

2.4 The Decibel-Scale

A logarithmic scale allows visualization of a wider dynamic range than a linear scale. This 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, it is defined by

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

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}, and 0 dB represents this reference value. This reference can be a predefined value (e.g., 1 V or 1 mW), an input level, or the maximum value.

Spectral data are mostly presented in dB. We are often only interested in the relative variation between spectral components, choosing the reference value 0 dB as the maximum value in the data set. In this case, all other values in the spectrum or spectrogram are negative decibels. 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 and values are given with at most one decimal. Some important dB values are listed in Table 3.

Table 3. List of important dB-values. See definition inn (10)

dB	Amplitude ratio v/v_{ref}		Power ratio 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	$\frac{1}{10}$	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}

2.5 Spectra and Spectrograms

2.5.1 Harmonics and Periodic Signals

Any signal $x(t)$ over an interval T_0 can be written as a sum of complex exponentials where the frequencies f_k are the harmonics of the fundamental f_0, i.e., $f_k = kf_0 = k/T_0, k = 0, 1, 2, \ldots$,

$$x(t) = \sum_{k=-\infty}^{+\infty} a_k e^{j2\pi kt/T_0} \quad , \qquad\qquad f_0 = \frac{1}{T_0} \, . \tag{11}$$

The strength of the frequency components are given by the Fourier coefficients a_k, found by the following equation, shown in the textbook [1]

$$a_k = \frac{1}{T_0} \int_0^{T_0} x(t) e^{-j2\pi kt/T_0} \mathrm{d}t \, . \tag{12}$$

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 $\Delta f = f_0 = 1/T_0$. Good resolution in frequency, i.e., low Δf, requires a long observation time, a large T_0. The resolution in time Δt is the segment length, $\Delta t = T_0$. This gives the important relation between resolution in time and frequency,

$$\Delta f = f_0 = \frac{1}{T_0} \qquad\qquad \Delta t = T_0 \qquad\qquad \Delta f = \frac{1}{\Delta t} \, . \tag{13}$$

This shows how we must compromise between resolution in time Δt and resolution in frequency Δf, and how this is controlled by the observation time T_0.

2.5.2 Spectra, Fourier Coefficients and FFT

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 very efficiently by using the *Fast Fourier Transform* algorithm, FFT. This is available as the function `fft` in the SciPy module that is also called `fft`. The relation between the results returned from the FFT-algorithm and the Fourier coefficients a_k are summarised as

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.

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 $x[n]$.

4) The arrray of Fourier coefficients returned from FFT are arranged with the negative coefficients *after* the positive coefficients, i.e., the lower half of the returned array contains the positive frequency components, the upper half contains the negative coefficients.

   This seemingly strange arrangement can easily be rearranged to make it easier interpret. The helper function `fftreshape` reorganizes the results so the negative frequencies come first.

5) The frequencies f_k corresponding to the Fourier coefficients a_k are given by

$$f_k = \frac{kf_s}{N_{fft}} \, , \tag{14}$$

   where N_{fft} is the number of samples used in the FFT, and f_s is the sample rate.

   This equation is built into the helper function `fftfreq`, which returns the frequencies as a NumPy array. The frequencies are organized in the same order as from `fft`, and can be rearranged by `fftreshape`.

6) The FFT algorithm works by breaking down the number of points in prime factors and works best when 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 Chapter 8 in the textbook [1].

Example code for how to calculate the Fourier coefficients, scale and order them correctly, and plot the magnitude and phase is shown in Table 4.

Table 4. 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` returns the frequency vector corresponding to the Fourier coefficients, and `fftshift` puts the positive and negative coefficients in the correct order. The FFT algorithm requires the output to be scaled with the number of samples to get correct magnitudes for a_k. The last part plots the Fourier coefficients in two graphs, one for magnitude and one for phase. Note that the phase can take seemingly random values where the magnitude is close to zero, due to numerical noise.

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

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

plt.subplot(1, 2, 1)           # Subplot for magnitudes
plt.stem(f, np.abs(a))         # Magnitude of spectral components as stem-plot
plt.xlabel("Frequency␣[Hz]")
plt.ylabel("Magnitude")
plt.grid(True)

plt.subplot(1, 2, 2)           # Subplot for phase
plt.stem(f, np.angle(a))       # Phase of spectral components as stem-plot
plt.xlabel("Frequency␣[Hz]")
plt.ylabel("Phase␣[radians]")
plt.grid(True)
```

2.5.3 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 the time interval T_0 and sample rate f_s, making comparisons 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

$$P_{xx} = \begin{cases} a_0^2 & k = 0 \\ \frac{1}{2}|a_k|^2 & k > 0 \end{cases} \quad . \tag{15}$$

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 5. The calculation of the PSD is the first line only, the rest is for plotting the results.

2.5.4 Spectrogram

Changes in frequency contents of signal with time can be illustrated with a spectrogram, see Chapter 3-6 in the textbook [1]. The spectrogram is found by splitting the time signal into shorter segments of length T_0 and finding the PSD over each segment. The result is displayed on a two-dimensional intensity plot with time on the x-axis and frequency on the y-axis. Some comments on how to use spectrograms are

1) Python offers several versions of spectrogram. A version that is straightforward and simple to configure is found in SciPy, `scipy.signal.spectrogram`.

2) Spectrograms are calculated over short segments of the signal. The finite length of the time segments limits the frequency resolution in the spectrogram, see (13).

Table 5. 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
plt.figure(figsize = [16, 8])  # Define figure for plots

plt.subplot(2, 1, 1)           # Subplot for signal
plt.plot(t, x)                 # Signal x as function of time t
plt.xlabel("Time␣[s]")
plt.ylabel("Amplitude")
plt.grid(True)

plt.subplot(2, 1, 2)           # Subplot for power spectral density
plt.plot(f, pxx)               # Power spectral density
plt.xlabel("Time␣[s]")
plt.ylabel("PSD␣[V^2/Hz]")
plt.grid(True)
```

3) Spectrograms can be difficult to configure. Critical parameters are the length of segments, the frequency axis scale, and the intensity display scale. Different settings can create spectrograms that look different even if presenting the same data set.

4) The results are best displayed in decibels on a 2D intensity image. The recommended way to do this is by the Matplotlib function `pcolormesh(t, f, s, vmin=s_min)`, where `s` is the spectrogram as a 2D array, `t` the time vector, `f` the frequency vector, and `s_min` the minimum value to be shown on the plot.

   `pcolormesh` is very similar to the function `imshow`, but is easier to match with x- and y-axis data.

5) The spectrogram is a pseudo-colour plot, each value is mapped to a colour from a predefined colourmap. Matplotlib's default colourmap is `viridis`, other examples are `inferno`, `gray`, `hot`, and `cool`.

Example of code for plotting a spectrogram is shown in Table 6.

2.6 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 (12), the results are

$$\text{Square wave } x_s(t) \qquad a_k = \begin{cases} \frac{2}{j\pi k} & k = \pm 1, \pm 3, \pm 5, \dots \\ 0 & k = 0, \pm 2, \pm 4, \pm 6, \dots \end{cases} \tag{16a}$$

$$\text{Triangle wave } x_t(t) \qquad a_k = \begin{cases} \frac{4}{j\pi^2 k^2}(-1)^{(k-1)/2} & k = \pm 1, \pm 3, \pm 5, \dots \\ 0 & k = 0, \pm 2, \pm 4, \pm 6, \dots \end{cases} \tag{16b}$$

The even components ($k = \pm 2, \pm 4, \dots$) vanish for both these wave. This comes out of equation (12), but can also be seen from symmetry. Note that the coefficients for the square wave decrease as $1/k$, while the coefficients for the triangle wave decrease as $1/k^2$. Hence, the coefficients for the triangle wave decrease more rapidly than for the square wave, and a good approximation of a triangle wave needs fewer coefficients approximation of a square wave.

Table 6. Example code for plotting the spectrogram of a signal x sampled at sample rate t. 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, in decibels. The parameter detrend can 'detrend' the data by removing DC-components.

```
# Configure spectrogram
n_segment = 1024  # Segment length for fft, defines time and frequency resolution
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
plt.figure(figsize=(16, 8))  # Define figure for results

plt.pcolormesh(t, f, sx_db, vmin=s_min)  # Draw spectrogram image

plt.xlabel("Time [s]")          # Axis labels and scales
plt.ylabel("Frequency [Hz]")
plt.ylim(0, f_max)

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

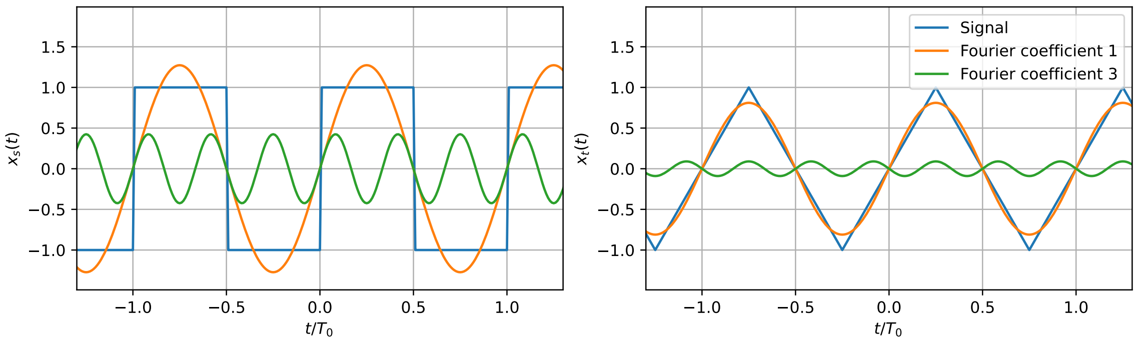

Figure 1. Square and triangle waves with the waves from the 1st and 3rd Fourier coefficients added. The triangle wave is better reproduced by a few Fourier coefficients than the square wave, and its Fourier coefficients decrease more rapidly as k increases.

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. If so, you can either copy and pate your code from Spyder into JupyterLab or load the Python-files into JupyterLab. This can then be exported to a 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. 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

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

Set the sample rate to exactly 32 samples per period.

b) A signal which is the sum of three sinusoids as in (3), described by

Signal component	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 $10\,000\,\mathrm{Samples/s}$ and duration to $0.1\,\mathrm{s}$.

Exercises

1) Plot the two signals $x(t)$ and their Fourier coefficients a_k. Use the code in Table 4 as template.
2) Plot the spectrogram of the two signals, using the code in Table 6 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

$$x(t) = A_1 \cos(2\pi(f_c - f_\Delta)t) + A_2 \cos(2\pi(f_c + f_\Delta)t)\,. \tag{17}$$

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

A template for the function header is given in Table 7. This can be made simple by calling the function `make_summed_cos` from Lab 1.

2) Test the function for the input values

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

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

Plot the signal on one subplot and its power spectrum in a second subplot. 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 $f_\Delta = 5\,\mathrm{Hz}$ and $2\,\mathrm{Hz}$. 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

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

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

6) Plot the spectrogram of this signal using the template from Table 6. 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 (13) in Section 2.5.1.

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

$$x(t) = \cos \Psi(t) \qquad \Psi(t) = 2\pi \mu t^2 + 2\pi f_0 t + \phi \, . \tag{18}$$

Specify the signal by start and end frequencies f_1 and f_2, phase ϕ, sample rate f_s, and duration. A template for the header of the function is given in Table 8.

2. Test the function for the input values

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

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

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 6. 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. Do they sound equal when played? 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

   ```
   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 $100\,\mathrm{Hz}$. 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 sawtooth wave functions in SciPy, `scipy.signal.square` and `scipy.signal.sawtooth`. Look in the documentation for SciPy to see how to configure them, especially, how to configure `sawtooth` to make a symmetric triangle wave.

2) Calculate and plot the Fourier coefficients of the two waves, use the template code in Table 4.

   Find the numerical values of the six lowest coefficients, a_k for $k = 0, 1, 2, \ldots 5$, from (16a) and (16b). Compare the plot with these values.

3) Calculate and plot the power spectrum of the two waves, use the template code in Table 5. 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 6. 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 $100\,\mathrm{Hz}$ and end at $4000\,\mathrm{Hz}$. Set the sample rate to $11\,025\,\mathrm{Hz}$, 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*, 2nd ed. United Kingdom: Pearson Education Limited, 2016.

[2] ——, “Lab P-4: AM and FM Sinusoidal Signals,” 2016. [Online]. Available: https://dspfirst.gatech.edu/database/?d=labs

[3] ——, “Lab S-8: Spectrograms: Harmonic Lines and Chirp Aliasing,” 2016. [Online]. Available: https://dspfirst.gatech.edu/database/?d=labs

[4] P. Raybaut. (2024) Spyder. [Online]. Available: https://docs.spyder-ide.org/

[5] Project Jupyter. (2024) Jupyter Lab. [Online]. Available: https://docs.jupyter.org

[6] NumPy. (2024) NumPy ver. 2.2. [Online]. Available: https://numpy.org/doc/2.2

[7] C. R. Harris et al., “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[8] Matplotlib. (2024) Matplotlib 3.10.0. [Online]. Available: https://matplotlib.org/stable

[9] J. D. Hunter, “Matplotlib: A 2D graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.1109/MCSE.2007.55

[10] SciPy. (2025) SciPy ver. 1.15. [Online]. Available: https://docs.scipy.org/doc/scipy

[11] P. Virtanen et al., “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020. [Online]. Available: https://doi.org/10.1038/s41592-019-0686-2