Exercise 13.1 Sampling length and sampling frequency

The Python code for this exercise is given in Listing 1.

(a) The different signals and the amplitude spectra can be found in Fig. 1. Comparing the amplitude spectra, we see that when we increase the sampling length (i.e. when we sample longer signals) we get a better resolution in *frequency*. The frequency resolution, Δf, is here directly related to the number of time samples in the signal, N, and the time step, Δt: $\Delta f = (\Delta t \times N)^{-1}$.

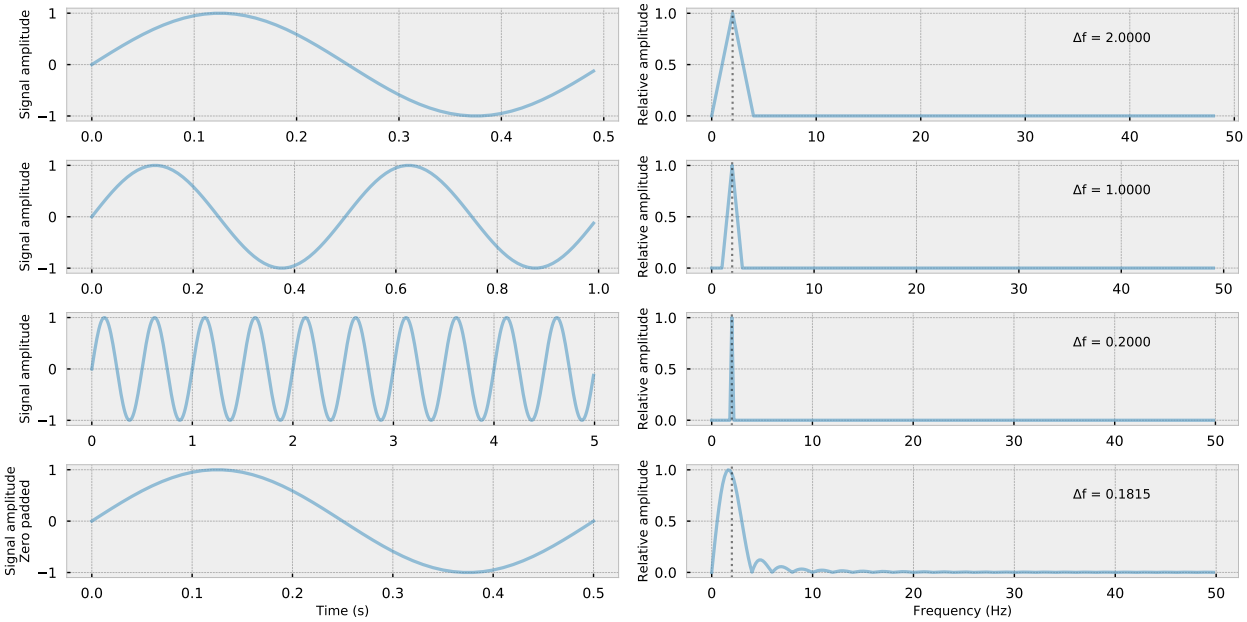

Figure 1: *The amplitude spectra of signals evaluated on different time intervals, with the same resolution in time. The frequency resolution, Δf, is indicated in the figure. The bottom signal is a zero-padded version of the top signal.*

(b) When we pad the signal, we are effectively increasing its length. We then expect that Δf should decrease, and we might expect that we get a better frequency resolution. The amplitude spectrum of the padded function can be found in Fig. 1, and indeed, we see that Δf has decreased. But, we do not really get a better "peak"-resolution. That is, if two peaks were close together, our ability to distinguish between them would not improve simply by padding the original signal. The second-last amplitude spectrum in Fig. 1 has a much better resolution (i.e. the peak is much narrower) even though the Δf is larger here, compared to the padded signal. The reason for this is that padding a signal does not add any new information to the signal, and the effect is that we are effective interpolating between the frequencies in the spectrum so that it appears smoother.

(c) The 4 signals and their amplitude spectra are shown in Fig. 2. Comparing the two first

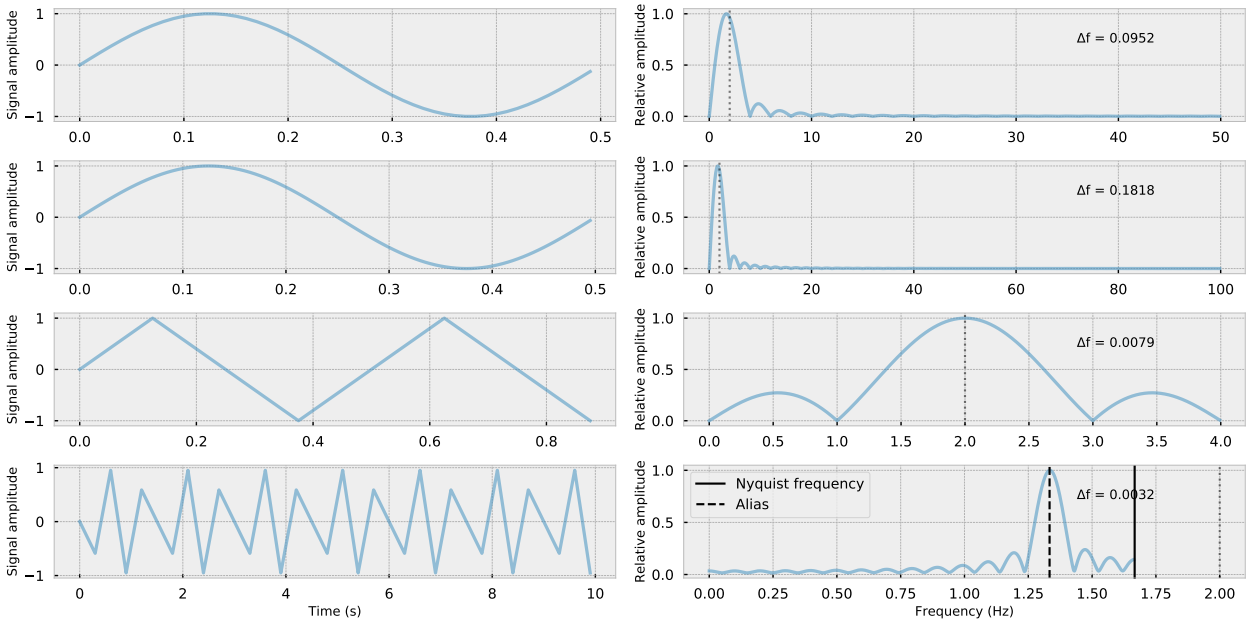

Figure 2: *The amplitude spectra of signals evaluated for different resolution in time and different lengths. The frequency resolution, Δf, is indicated in the figure. For the last signal, we have highlighted the Nyquist frequency and the observed alias frequency.*

signals, we see that the Δf increases when we decrease Δt, as expected. We also note that the maximum frequency we can obtain also increases. The maximum frequency, $f_{\max}$, we can obtain is inversely related to Δt: $f_{\max} = (\Delta t)^{-1}$. So, measuring the signal more often means that we can resolve higher frequencies.

From the Nyquist–Shannon sampling theorem, we know that if a function contains no frequencies higher than B Hz, it is completely determined by measuring it at a series of points spaced $1/(2B)$ seconds apart. For this particular signal, with a frequency of 2 Hz, this rate is $1/(2 \times 2\,\text{Hz}) = 0.25$ s. For the first of these two signals we used a time sampling of $0.125\,\text{s} < 0.25$ s and we can thus resolve the frequency of 2 Hz as shown in the second last amplitude spectrum in Fig. 2. In the last signal, we used a time sampling of $0.3\,\text{s} > 0.25$ s and we can no longer resolve the frequency of 2 Hz. In this figure, we also show the Nyquist frequency which is half of the sampling rate, $0.5 \times \frac{1}{0.3\text{s}} = 1.67$ Hz. Here, this will be the highest frequency we can resolve and higher frequencies will be aliased. This is exactly what we see in Fig. 2. Rather than seeing a peak for 2 Hz, we see a peak at $1.67\,\text{Hz} - (2\,\text{Hz} - 1.67\,\text{Hz}) = 1.33$ Hz, as we have not measured the signal often enough to resolve the 2 Hz peak.

(d) From what we have seen so far, we can now estimate how frequently we have to measure the given signal to resolve the peaks. First of all, the highest frequency is 2.1 Hz, so

we should at maximum have a time step of $1/(2 \times 2.1, \mathrm{Hz}) = 0.238\,\mathrm{s}$. Let us pick a time step of $\Delta t = 0.01$ s to be very conservative. Further, the frequency difference between the two frequencies are $2.1 - 2 = 0.1$ Hz, so we need to sample the signal for a length that gives a $\Delta f < 0.1$ Hz. To be sure that we can distinguish the peaks, we set $\Delta f = 0.1\,\mathrm{Hz}/4 = 0.025$ Hz. Since we have selected a Δt, we can now estimate the length of the signal: $N > \frac{1}{0.025\Delta t} = 4000$. We test this out in practice in Fig. 3. Here,

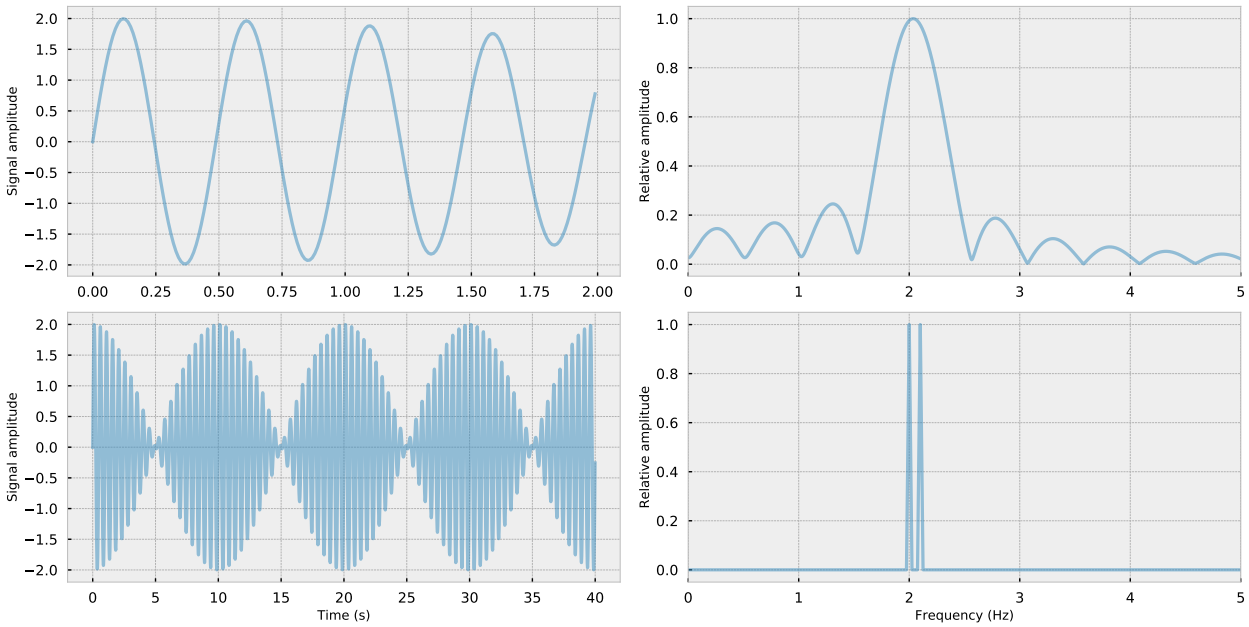

Figure 3: *The amplitude spectra of the signal* $\sin(2 \times 2\pi t) + \sin(2.1 \times 2\pi t)$. *The top figure shows the obtained spectrum when sampling for only 2 s. The bottom figure shows the spectrum when sampling for 40 s. In the latter case, we can identify the two frequencies 2 Hz and 2.1 Hz in the amplitude spectrum.*

we show two versions of the signal. The first is sampled for a short time and we get a poor frequency resolution and we only see a single peak. The second version uses a longer time sampling, and we see here that we can resolve the two frequencies in the signal.

Exercise 13.2 Finding a signal in a noisy experiment

The Python code for this exercise is given in Listing 2. When calculating the FFT of the given signal, we find the four largest amplitudes as given in Table 1. Based on this, we conclude that the signal should have the following form,

$$y(t) = 2.554 \sin(26 \times 2\pi t) + 3.84 \sin(7 \times 2\pi t). \tag{1}$$

Amplitude	Frequency
1.277	−26.0
1.277	26.0
1.928	7.0
1.928	−7.0

Table 1: *The four largest amplitudes and corresponding frequencies found in the given signal*

This equation is shown, together with the original signal and the amplitude spectrum in Fig 4. Our estimated signal compares well with the actual signal used to create the raw data which was,

$$y(t) = 2.5\sin(26 \times 2\pi t) + 4\sin(7 \times 2\pi t). \tag{2}$$

Exercise 13.3 Finding a note

The Python code for this exercise is given in Listing 3.

(a) The amplitude spectrum of the given signal is shown in Fig. 5. We see here that we find only one frequency in the signal, 440 Hz, as expected.

(b) (i) The amplitude spectrum of the first part of the signal is shown in Fig. 6. The locations of the three largest peaks are also summarized in table 2. We see that this part of the signal mainly contains the note F as expected. We note that there is an offset compared to tabulated* frequencies of the F note. In general, it is hard to say if a note is out of tune when we are comparing frequencies that are close together. This depends on the person listening in addition to the frequency itself. If two notes are played at the same time, it is easier to note if one is out of tune as we then can actually hear the interference pattern between the two notes as beats. Wikipedia gives a Just-noticeable difference of 3 Hz for frequencies below 500 Hz for simple tones, and based on this we can conclude that these notes will be perceived as the F note. If you want to create sound files with specific frequencies, Listing 4 gives a Python code for creating sound files (wav) where you can specify frequencies. You can test this Python code for the frequencies we have considered here, and see if you can hear any difference.

I can not distinguish the note played in this signal from a F note. Maybe some of you can?

(ii) The amplitude spectrum for the second part of the signal is given in Fig. 7. The locations of the four largest peaks are also summarized in table 3. We see that

*https://pages.mtu.edu/~suits/notefreqs.html

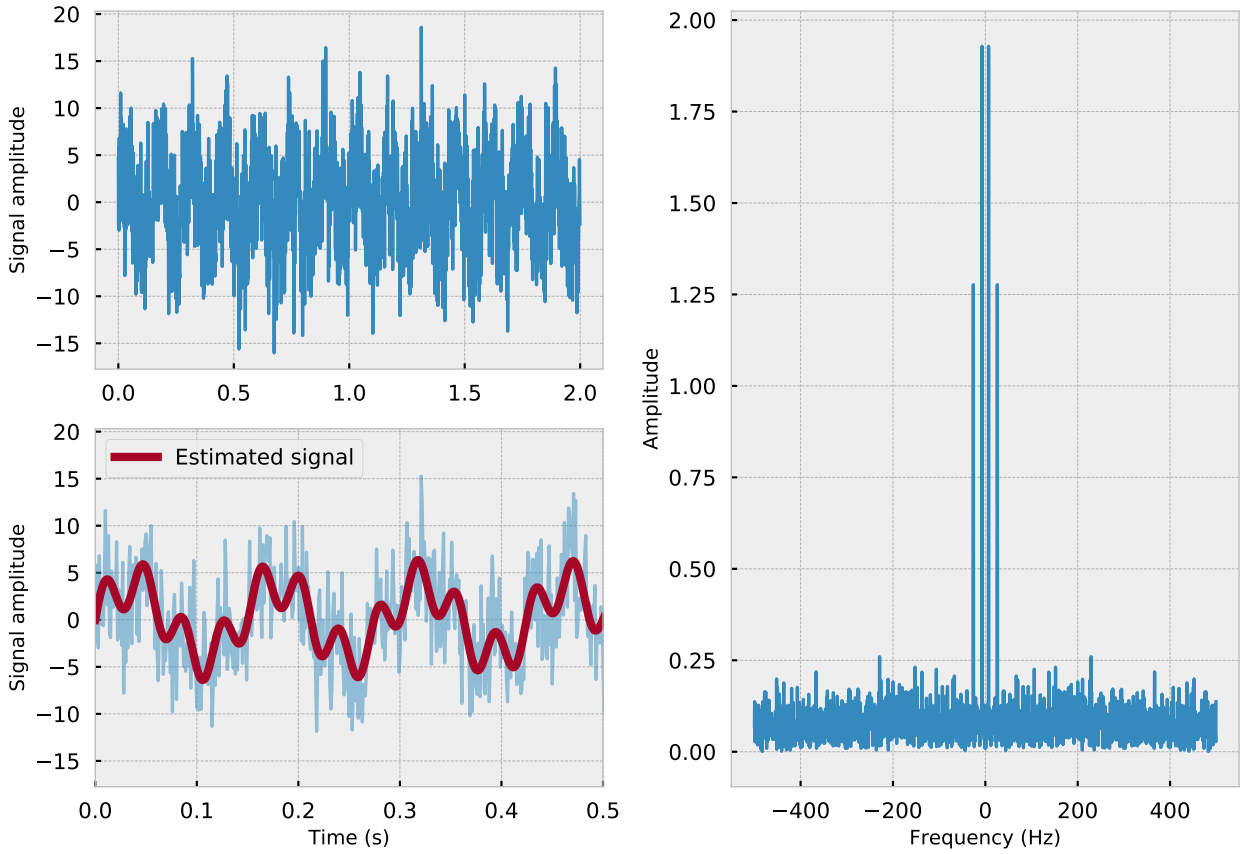

Figure 4: *(Top-left) The measured signal. (Bottom-left) The measured signal and the estimated signal without noise. (Right) The amplitude spectrum of the measured signal.*

Frequency	Closest note	Offset in cents
173.85	F_3 (174.61 Hz)	7.53
347.70	F_4 (349.23 Hz)	7.58
696.95	F_5 (698.46 Hz)	3.75

Table 2: *The three largest amplitudes found for part 1 of the signal. The offsets in cents are calculated from the ratio of the frequencies.*

this part of the signal mainly contain the note F, in addition to a note close to the A note. Again, we see that all the notes are slightly off compared to the tabulated frequencies. If we again use the criterion of a difference of 3 Hz below 500 Hz, then we expect to not be able to hear that the F notes are out of tune, but the note close to A at 436.7 Hz could be perceived as being out of tune.

(iii) The spectra are compared in Fig. 8. We see here that the main difference between

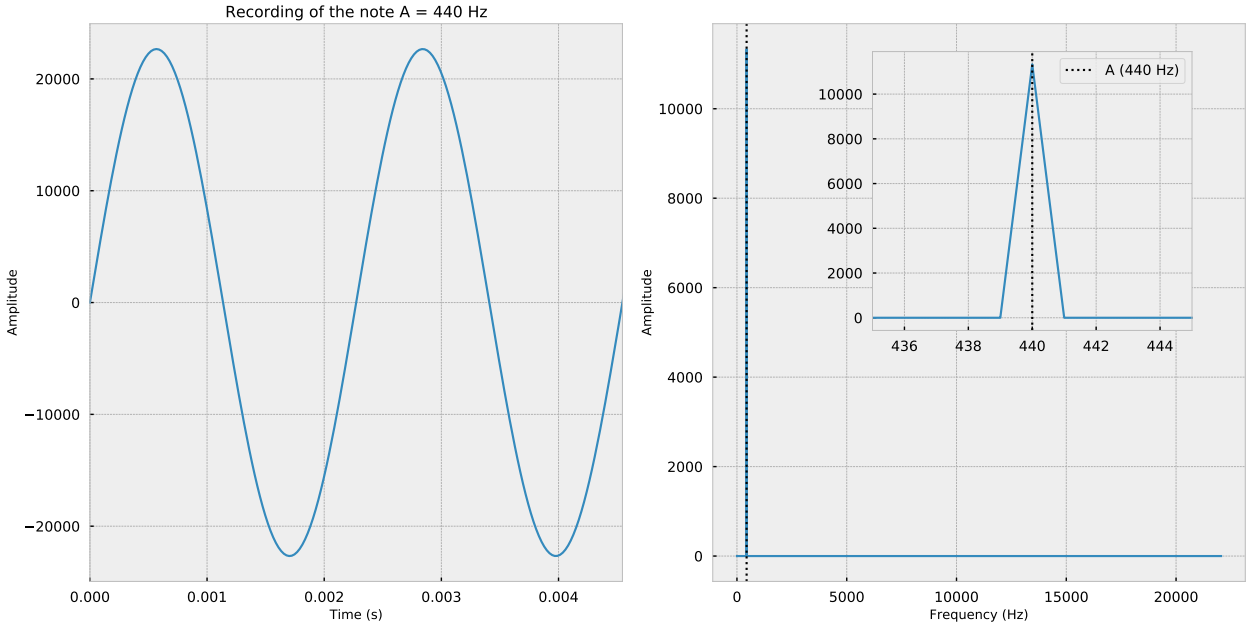

Figure 5: *(Left) The measured signal. (Right) The amplitude spectrum of the measured signal. The inset shows the spectrum in a small range around* 440 *Hz.*

Frequency	Closest note	Offset in cents
173.35	F_3 (174.61 Hz)	12.5736
346.69	F_4 (349.23 Hz)	12.6232
436.70	A_4 (440.00 Hz)	13.0345
696.72	F_5 (698.46 Hz)	4.31989

Table 3: *The three largest amplitudes found for part 2 of the signal. The offsets in cents are calculated from the ratio of the frequencies.*

the two signals is the note at 436.7 Hz. Putting together what we know about the parts, we conclude that this frequency is the second note played in the intro.

(c) Based on the cents given in Table 3, and the criterion of a difference of 3 Hz as being perceivable as out of tune, we conclude that the string playing the second note in the intro is slightly out of tune. Here is a youtube video explaining why the intro still sounds nice.

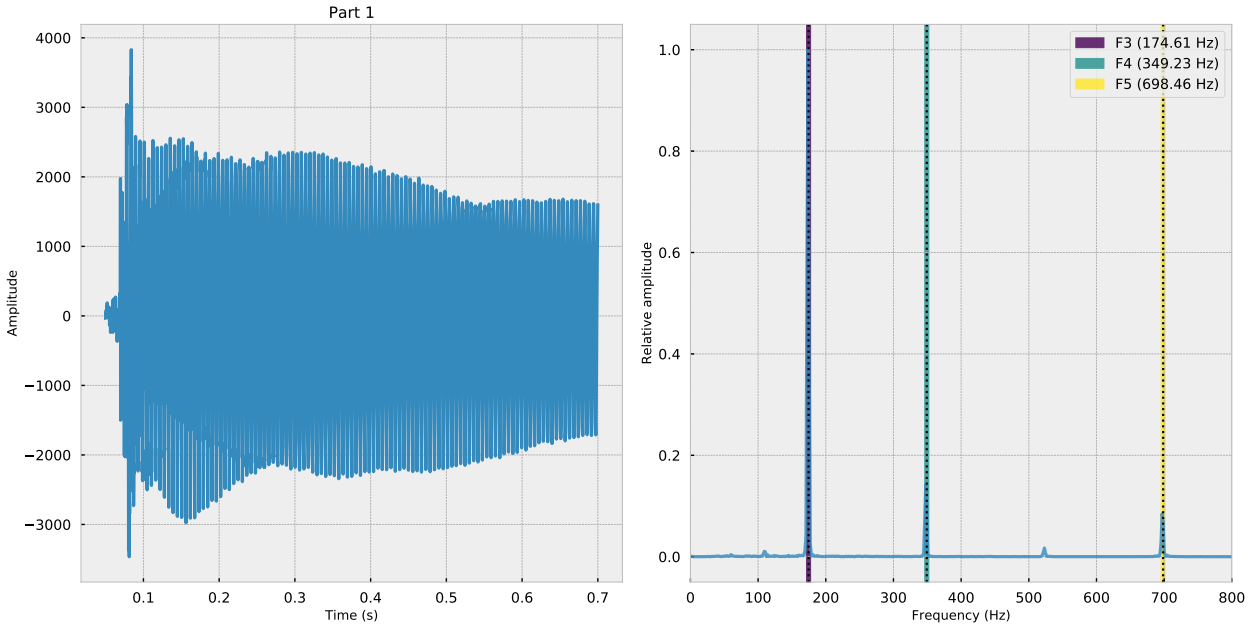

Figure 6: *(Left) The measured signal. (Right) The amplitude spectrum of the measured signal. Frequencies for a selection of notes have been highlighted in the figure and the highlighted regions corresponds to* ± 3 *Hz around the relevant note.*

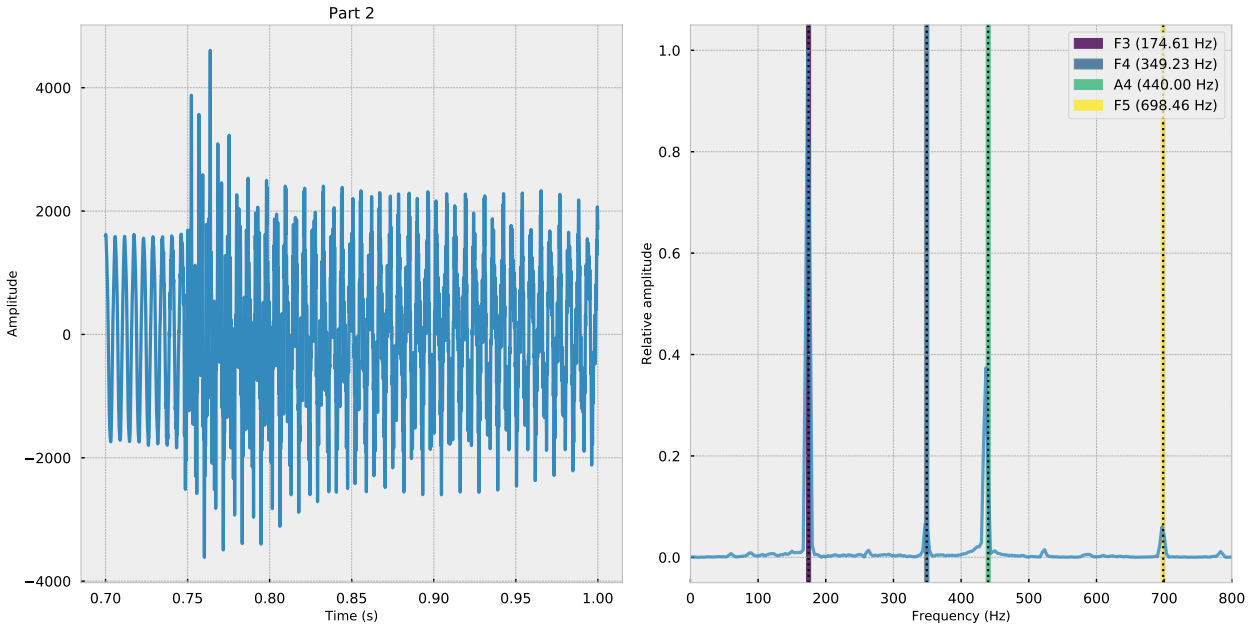

Figure 7: *(Left) The measured signal. (Right) The amplitude spectrum of the measured signal. Frequencies for a selection of notes have been highlighted in the figure and the highlighted regions corresponds to* ± 3 *Hz around the relevant note.*

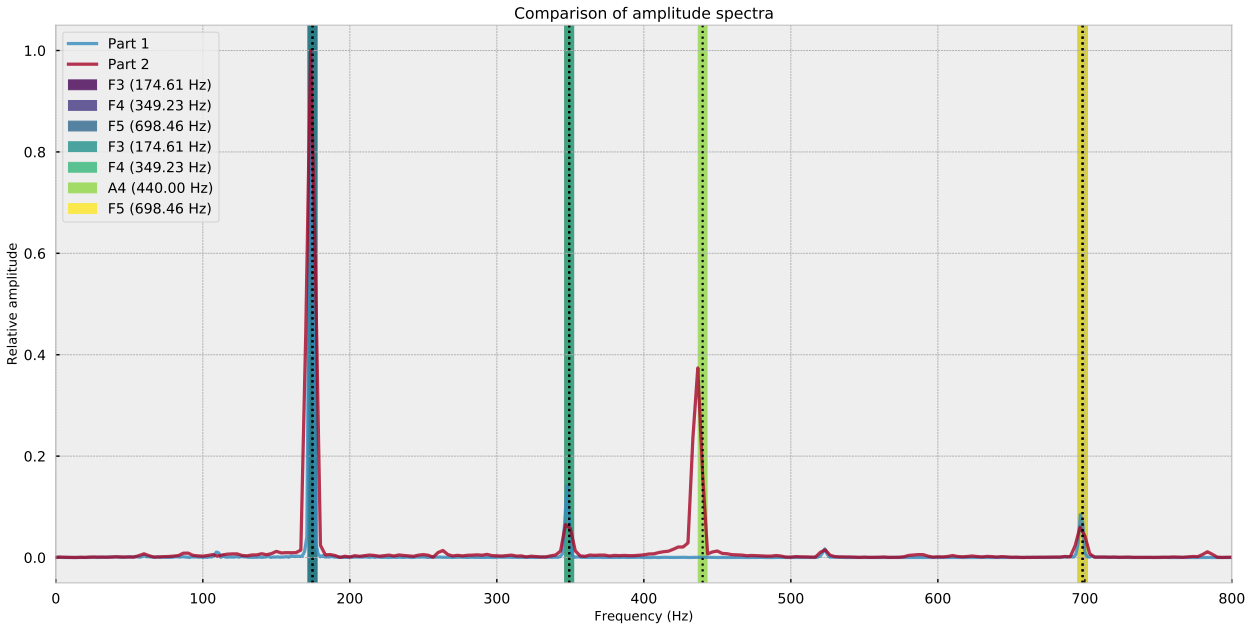

Figure 8: *Comparison of the amplitude spectra for the two parts of the measured signal. Frequencies for a selection of notes have been highlighted in the figure and the highlighted regions corresponds to* ± 3 *Hz around the relevant note.*

Python solutions

```
import numpy as np
from scipy.fft import fft, fftfreq
import matplotlib as mpl
from matplotlib import pyplot as plt
mpl.rcParams["savefig.format"] = 'pdf'
plt.style.use(['seaborn-talk', 'bmh'])


def run_fft(time, signal):
    """Do fft for the given signal."""
    fourier = fft(signal)
    delta_t = time[1] - time[0]
    N = signal.size
    freq = fftfreq(N, delta_t)
    amplitude = np.abs(fourier / N)
    power = np.abs(fourier)**2
    # Normalize power:
    power = power / power.sum()
    return freq[:N//2], fourier, amplitude[:N//2], power[:N//2]


def part1():
    """Investigate the effect of increating the singal length."""
    delta_t = 0.01
    freq_0 = 2.0
    tmax = 1.0/freq_0

    periods = [1, 2, 10]

    fig, axes = plt.subplots(
        constrained_layout=True, ncols=2,
        nrows=len(periods)+1,
        figsize=(16, 8)
    )
    for i, peri in enumerate(periods):
        time = np.arange(0, peri * tmax, delta_t)
        signal = np.sin(freq_0 * 2.0 * np.pi * time)
        freq, _, amplitude, _ = run_fft(time, signal)

        axes[i, 0].plot(time, signal, alpha=0.5, lw=3)
        axes[i, 0].set_ylabel('Signal amplitude')
        axes[i, 1].plot(freq, amplitude / amplitude.max(), alpha=0.5, lw=3)
        axes[i, 1].text(0.7, 0.7,
                        'Δf = {:.4f}'.format(freq[1]-freq[0]),
                        fontsize='large',
                        transform=axes[i, 1].transAxes)
        axes[i, 1].set_ylabel('Relative amplitude')
        axes[i, 1].axvline(x=freq_0, ls=':', color='k', alpha=0.5)
```

```
    time = np.arange(0, tmax + delta_t, delta_t)
    signal = np.sin(freq_0 * 2.0 * np.pi * time)
    zero_pad = np.pad(signal, pad_width=250)
    freq, _, amplitude, _ = run_fft(time, zero_pad)
    last_ax = axes[-1]
    last_ax[0].plot(time, signal, alpha=0.5, lw=3)
    last_ax[0].set_ylabel('Signal amplitude\nZero padded')
    last_ax[1].plot(freq, amplitude / amplitude.max(), alpha=0.5, lw=3)
    last_ax[1].set_ylabel('Relative amplitude')
    last_ax[1].axvline(x=freq_0, ls=':', color='k', alpha=0.5)
    last_ax[0].set_xlabel('Time (s)')
    last_ax[1].set_xlabel('Frequency (Hz)')
    last_ax[1].text(
        0.7, 0.7,
        'Δf = {:.4f}'.format(freq[1]-freq[0]),
        fontsize='large',
        transform=last_ax[1].transAxes
    )
    fig.savefig('fft1', bbox_inches='tight')


def part2():
    """Investigate the effect of changing sampling frequency."""
    freq_0 = 2
    tmax = 1.0/freq_0

    delta_ts = [0.01, 0.005, 0.125, 0.3]
    periods = [1, 1, 2, 20]

    fig, axes = plt.subplots(
        constrained_layout=True, ncols=2, nrows=len(delta_ts),
        figsize=(16, 8)
    )

    for i, delta_t in enumerate(delta_ts):
        time = np.arange(0, periods[i]*tmax, delta_t)
        signal = np.sin(freq_0 * 2.0 * np.pi * time)
        zero_pad = np.pad(signal, pad_width=500)
        freq, _, amplitude, _ = run_fft(time, zero_pad)
        axes[i, 0].plot(time, signal, alpha=0.5, lw=3)
        axes[i, 0].set_ylabel('Signal amplitude')
        axes[i, 1].plot(freq, amplitude / amplitude.max(), alpha=0.5, lw=3)
        axes[i, 1].text(0.7, 0.7,
                        'Δf = {:.4f}'.format(freq[1]-freq[0]),
                        fontsize='large',
                        transform=axes[i, 1].transAxes)
        axes[i, 1].set_ylabel('Relative amplitude')
        axes[i, 1].axvline(x=freq_0, ls=':', color='k', alpha=0.5)
```

```
    axes[-1, 0].set_xlabel('Time (s)')
    axes[-1, 1].set_xlabel('Frequency (Hz)')
    f_nyquist = 0.5*1.0/delta_ts[-1]
    axes[-1, 1].axvline(x=f_nyquist, ls='-', color='k',
                        label='Nyquist frequency')
    fold = f_nyquist - (freq_0 - f_nyquist)
    axes[-1, 1].axvline(x=fold, ls='--', color='k',
                        label='Alias')
    axes[-1, 1].legend()
    fig.savefig('fft2', bbox_inches='tight')


def part3():
    """Here we test out if we can separate two signals close in frequency."""
    freq_0 = 2
    freq_1 = 2.1

    delta_t = 0.01

    fig, axes = plt.subplots(constrained_layout=True, ncols=2, nrows=2,
                             figsize=(16, 8))
    # 1) Use a relatively short time + pad it
    time = np.arange(0, 2, delta_t)
    signal = (
        np.sin(freq_0 * 2.0 * np.pi * time) +
        np.sin(freq_1 * 2.0 * np.pi * time)
    )
    zero_pad = np.pad(signal, pad_width=10000)  # Pretty hefty pad.
    freq, _, amplitude, _ = run_fft(time, zero_pad)
    axi = axes[0]
    axi[0].plot(time, signal, alpha=0.5, lw=3)
    axi[1].plot(freq, amplitude / amplitude.max(), alpha=0.5, lw=3)
    axi[1].set_xlim(0, 5)

    # 2) Use a longer time so that the frequency resolution is 0.025:
    time = np.arange(0, 40, delta_t)
    signal = (
        np.sin(freq_0 * 2.0 * np.pi * time) +
        np.sin(freq_1 * 2.0 * np.pi * time)
    )
    freq, _, amplitude, _ = run_fft(time, signal)
    axi = axes[1]
    axi[0].plot(time, signal, alpha=0.5, lw=3)
    axi[1].plot(freq, amplitude / amplitude.max(), alpha=0.5, lw=3)
    axi[1].set_xlim(0, 5)

    for axi in axes:
        axi[0].set_ylabel('Signal amplitude')
        axi[1].set_ylabel('Relative amplitude')
    axes[-1, 0].set_xlabel('Time (s)')
```

```
    axes[-1, 1].set_xlabel('Frequency (Hz)')
    fig.savefig('fft3', bbox_inches='tight')


def main():
    part1()
    part2()
    part3()
    plt.show()


if __name__ == '__main__':
    main()
```

Listing 1: *Python code for investigating sampling length and frequency (exercise 13.1).*

```
import numpy as np
from scipy.fft import fft, fftfreq, fftshift
import matplotlib as mpl
from matplotlib import pyplot as plt
mpl.rcParams["savefig.format"] = 'pdf'
plt.style.use(['seaborn-talk', 'bmh'])


def run_fft(time, signal):
    """Do fft for the given signal."""
    fourier = fft(signal)
    delta_t = time[1] - time[0]
    N = signal.size
    freq = fftfreq(N, delta_t)
    amplitude = np.abs(fourier / N)
    power = np.abs(fourier)**2
    phase = np.angle(fourier)
    # Normalize power:
    power = power / power.sum()
    return freq, fourier, amplitude, power, phase, N


def main():
    data = np.loadtxt('Data/unknown_signal.txt')
    time, signal = data[:, 0], data[:, 1]
    freq, fourier, amplitude, _, phase, N = run_fft(time, signal)

    idx = np.argsort(amplitude)
    for i in idx[-4:]:
        print(amplitude[i], fourier[i], phase[i] / np.pi, freq[i])

    fig = plt.figure(constrained_layout=True)
    grid = fig.add_gridspec(2, 2)
    ax1 = fig.add_subplot(grid[0, 0])
    ax2 = fig.add_subplot(grid[1, 0])
```

```
    ax3 = fig.add_subplot(grid[:, 1])
    ax1.plot(time, signal)
    ax2.plot(time, signal, alpha=0.5)

    ax3.plot(fftshift(freq), fftshift(amplitude))
    ax3.set(xlabel='Frequency (Hz)', ylabel='Amplitude')
    ax1.set_ylabel('Signal amplitude')
    ax2.set_ylabel('Signal amplitude')
    ax2.set_xlabel('Time (s)')

    signal_0 = (
        2 * 1.9281376233623388 * np.sin(7 * 2.0 * np.pi * time) +
        2 * 1.276876378320796 * np.sin(26 * 2 * np.pi * time)
    )
    ax2.plot(time, signal_0, lw=5, label='Estimated signal')
    ax2.set_xlim(0, 0.5)
    ax2.legend()
    fig.savefig('signal', bbox_inches='tight')
    plt.show()


if __name__ == '__main__':
    main()
```

Listing 2: *Python code for finding the coefficients for the known signal (exercise 13.2).*

```
import numpy as np
from scipy.fft import fft, fftfreq
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib.cm import get_cmap
from scipy.signal import find_peaks
import pandas as pd
from tabulate import tabulate
mpl.rcParams["savefig.format"] = 'pdf'
plt.style.use(['seaborn-talk', 'bmh'])


def generate_colors(listlike, cmap_name='spring'):
    """Generate n colors."""
    return get_cmap(name=cmap_name)(np.linspace(0, 1, len(listlike)))


def run_fft(time, signal):
    """Do fft for the given signal."""
    fourier = fft(signal)
    delta_t = time[1] - time[0]
    N = signal.size
    freq = fftfreq(N, delta_t)
    amplitude = np.abs(fourier / N)
    power = np.abs(fourier)**2
```

```
    # Normalize power:
    power = power / power.sum()
    return freq[:N//2], fourier, amplitude[:N//2], power[:N//2]


def calculate_cents(freq1, freq2):
    """Return the cents for a frequency ratio."""
    return 1200. * np.log2(max((freq1, freq2)) / min((freq1, freq2)))


def get_freq_from_cents(freq, cents):
    """Return a range around a frequency for a given number of cents."""
    freqi = freq * 2**(cents / 1200.)
    diff = np.abs(freq - freqi)
    return freq - diff, freq + diff


def find_closest_note(notes, freq):
    """For a given frequency, find the closest note."""
    note = notes.iloc[(notes['Frequency']-freq).abs().argmin()]
    note_c, freq_c = note.values
    return note_c, freq_c, calculate_cents(freq, freq_c)


def investigate_440():
    """Investigate the 440 hz signal."""
    data = np.loadtxt('Data/440hz.txt.gz')
    time, signal = data[:, 0], data[:, 1]
    freq, _, amplitude, _ = run_fft(time, signal)
    fig, (ax1, ax2) = plt.subplots(constrained_layout=True, ncols=2,
                                   figsize=(16, 8))
    ax3 = ax2.inset_axes([0.30, 0.45, 0.6, 0.5])

    ax1.plot(time, signal)
    ax1.set_xlim(0, 2/440.0)
    ax1.set_title('Recording of the note A = 440 Hz')
    ax1.set(xlabel='Time (s)', ylabel='Amplitude')
    ax2.plot(freq, amplitude)
    ax2.set(xlabel='Frequency (Hz)', ylabel='Amplitude')
    ax3.plot(freq, amplitude)
    ax3.axvline(x=440, label='A (440 Hz)', ls=':', color='k')
    ax2.axvline(x=440, label='A (440 Hz)', ls=':', color='k')
    ax3.set_xlim(435, 445)
    ax3.legend()
    fig.savefig('440hz', bbox_inches='tight')


def investigate_part1(notes):
    """Investigate part1 of the signal."""
    data = np.loadtxt('Data/part1.txt.gz')
```

```
    time, signal = data[:, 0], data[:, 1]
    window = np.hanning(len(signal))
    window /= window.sum()
    freq, _, amplitude, _ = run_fft(time, signal * window)

    amplitude /= amplitude.max()

    fig, (ax1, ax2) = plt.subplots(constrained_layout=True, ncols=2,
                                   figsize=(16, 8))
    ax1.plot(time, signal, lw=3)
    ax1.set_title('Part 1')
    ax1.set(xlabel='Time (s)', ylabel='Amplitude')
    ax2.plot(freq, amplitude, lw=3, alpha=0.8)
    ax2.set(xlabel='Frequency (Hz)', ylabel='Relative amplitude')

    # Create a table and add some plot annotations:
    peaks = find_peaks(amplitude, distance=3, threshold=0.01)[0]
    print('\nLargest peaks (part1):\n')
    headers = ['Frequency', 'Closest note', 'Offset in cents']
    table = []
    colors = generate_colors(peaks, 'viridis')
    for i, color in zip(peaks, colors):
        note, freq_note, cents = find_closest_note(notes, freq[i])
        table.append(
            [
                freq[i],
                '{0:} ({1:.2f} Hz)'.format(note, freq_note),
                cents,
            ]
        )
        ax2.axvline(x=freq_note,
                    ls=':', color='k')

        ax2.axvspan(xmin=freq_note-3, xmax=freq_note+3,
                    alpha=0.8,
                    label='{0:} ({1:.2f} Hz)'.format(note, freq_note),
                    color=color)

    print(tabulate(table, headers=headers, tablefmt='simple'))
    ax2.set_xlim(0, 800)
    ax2.legend()
    fig.savefig('part1', bbox_inches='tight')
    return freq, amplitude, [freq[i] for i in peaks]


def investigate_part2(notes):
    """Investigate part2 of the signal."""
    data = np.loadtxt('Data/part2.txt.gz')
    time, signal = data[:, 0], data[:, 1]
    window = np.hanning(len(signal))
```

```
    window /= window.sum()
    freq, _, amplitude, _ = run_fft(time, signal * window)

    amplitude /= amplitude.max()

    fig, (ax1, ax2) = plt.subplots(constrained_layout=True, ncols=2,
                                   figsize=(16, 8))
    ax1.plot(time, signal, lw=3)
    ax1.set_title('Part 2')
    ax1.set(xlabel='Time (s)', ylabel='Amplitude')
    ax2.plot(freq, amplitude, lw=3, alpha=0.8)
    ax2.set(xlabel='Frequency (Hz)', ylabel='Relative amplitude')

    # Create a table and add some plot annotations:
    peaks_all = find_peaks(amplitude, distance=3, threshold=0.007)[0]
    peaks = [i for i in peaks_all if freq[i] < 800]
    print('\nLargest peaks (part2):\n')
    headers = ['Frequency', 'Closest note', 'Offset in cents']
    table = []
    colors = generate_colors(peaks, 'viridis')
    for i, color in zip(peaks, colors):
        note, freq_note, cents = find_closest_note(notes, freq[i])
        table.append(
            [
                freq[i],
                '{0:} ({1:.2f} Hz)'.format(note, freq_note),
                cents,
            ]
        )
        ax2.axvline(x=freq_note,
                    ls=':', color='k')

        ax2.axvspan(xmin=freq_note-3, xmax=freq_note+3,
                    alpha=0.8,
                    label='{0:} ({1:.2f} Hz)'.format(note, freq_note),
                    color=color)

    print(tabulate(table, headers=headers, tablefmt='simple'))
    ax2.set_xlim(0, 800)
    ax2.legend()
    fig.savefig('part2', bbox_inches='tight')
    return freq, amplitude, [freq[i] for i in peaks]


def main():
    notes = pd.read_csv('Data/notes.csv')
    investigate_440()
    freq1, amplitude1, peaks1 = investigate_part1(notes)
    freq2, amplitude2, peaks2 = investigate_part2(notes)
    peaks = peaks1 + peaks2
```

```
    fig, ax1 = plt.subplots(constrained_layout=True,
                            figsize=(16, 8))
    ax1.set_title('Comparison of amplitude spectra')
    ax1.plot(freq1, amplitude1, lw=3, alpha=0.8, label='Part 1')
    ax1.plot(freq2, amplitude2, lw=3, alpha=0.8, label='Part 2')
    ax1.set(xlabel='Frequency (Hz)', ylabel='Relative amplitude')
    ax1.set_xlim(0, 800)

    colors = generate_colors(peaks, 'viridis')
    for freq, color in zip(peaks, colors):
        note, freq_note, cents = find_closest_note(notes, freq)
        ax1.axvline(x=freq_note,
                    ls=':', color='k')
        ax1.axvspan(xmin=freq_note-3, xmax=freq_note+3,
                    alpha=0.8,
                    label='{0:} ({1:.2f} Hz)'.format(note, freq_note),
                    color=color)
    ax1.legend()
    fig.savefig('comparison', bbox_inches='tight')
    plt.show()


if __name__ == '__main__':
    main()
```

Listing 3: *Python code for finding a note (exercise 13.3).*

```
import numpy as np
from scipy.io import wavfile


def create_test_sound(frequencies, length, outfile='sound.wav'):
    """Create a test sound for given frequencies and length in seconds."""
    sample_rate = 44100
    time = np.linspace(0, length, sample_rate * length)
    signal = np.zeros_like(time)
    for freq in frequencies:
        signal += 0.1 * np.sin(freq * 2 * np.pi * time)
    wavfile.write(outfile, sample_rate, signal)


def main():
    create_test_sound([173.852], 2, outfile='173.852hz.wav')
    create_test_sound([174.61], 2, outfile='174.61hz.wav')
    create_test_sound([173.852, 174.61], 2, outfile='173.852+174.61hz.wav')
    create_test_sound([436.7], 2, outfile='436.7hz.wav')
    create_test_sound([440], 2, outfile='440hz.wav')
    create_test_sound([436.7, 440], 2, outfile='436.7+440hz.wav')
    # Create a G chord:
    create_test_sound(
```

```
        [196.00, 246.94, 146.83],  # G3, B3, D3
        2,
        outfile='gchord.wav',
    )


if __name__ == '__main__':
    main()
```

Listing 4: *Python code for creating wav files with specified frequencies.*