Exercise 1: An introduction to Digital Signal Processing using Python

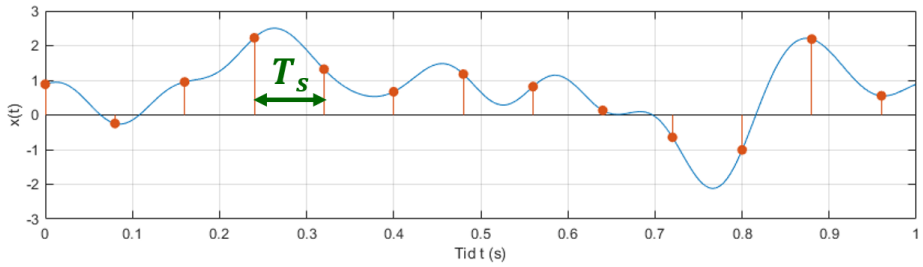
Kai Erik

August 15, 2025

In this exercise, the focus is on audio signals. The exercise problems cover topics from sampling and audio file formatting, to spectral analysis of digital audio and simple filtering of audio signals.

1 Exploring an audio file

When audio signals are recorded to a digital medium, this is done by sampling the amplitude of an analog signal (aka. voltage output from a microphone) at regular intervals known as the *sampling period* T_s, resulting in a sequence of measured values. An illustration of this process is shown below.



Closely related to the sampling period T_s is the *sampling rate* or *sampling frequency* f_s, defined as the *number of samples per second*. The sampling frequency f_s is calculated simpy by inverting the samping period T_s.

$$f_s = \frac{1}{T_s}$$

In order to reconstruct an analog voltage signal which can be sent to speakers or headphones, both knowledge of the sampling frequency used during recording *and* the measured sample values are required. Happily, all this information is contained within any audio file format.

The codecell below uses a function available through the `scipy` module to read the audio file `example_music.wav` containing some relaxing classical music. The sequence of audio amplitude measurements which contain the actual audio signal is assigned to the variable `sampleData` in the form of a very long `array`, while the sampling frequency is an integer value which is assigned to the variable `fs`.

After loading the audio file, the data is passed along to an embedded Auduio playback widget accessible through the IPython module.

[2]:
```
import scipy.io.wavfile as wavfile # Import module for handling of .wav audio␣
↪files
from IPython.display import Audio   # For loading embedded audio player

fs, sampleData = wavfile.read("sample_audio.wav") # "fs" is sampling frequency,␣
↪"sampleData" is the sequence of measurements

# Use the following lines to listen to the audio signal
Audio(sampleData, rate=fs)
```

[2]:
```
<IPython.lib.display.Audio object>
```

1.1 a)

Run the codecell and listen to the audio clip. We can now begin our analysis of the audio signal.

Using the data gathered in the codecell above, write a script which prints the sampling frequency f_s, sampling period T_s and the total duration of the audio file. *extra: what is the size of the file in number of bits? hint: the function* `len()` *should be useful here.*

[3]:
```
# BEGIN SOLUTION
print("Sample rate:", fs, "samples per second")
print("Sample period:", 1/fs*1e6, "microseconds")
print("Duration of audio file:", len(sampleData)/fs, "seconds")
# END SOLUTION
```

```
Sample rate: 22050 samples per second
Sample period: 45.35147392290249 microseconds
Duration of audio file: 20.340725623582767 seconds
```

1.2 b)

Use the Audio object to play back the audio clip once with double the sample rate f_s, and once with half the sample rate. What effects can you hear? Explain the cause of these effects.

[4]:
```
### BEGIN SOLUTION
Audio(sampleData, rate=fs*2) # Playback at 2X sample rate
#Audio(sampleData, rate=fs/2) # Playback at 0.5X sample rate
### END SOLUTION
```

[4]:
```
<IPython.lib.display.Audio object>
```

- Doubling the sample rate speeds up the playback by a factor of two. All sounds have higher pitch.
- Halving the sample rate slows down the playback by a factor of two. All sounds have lower pitch.

A crucial parameter to be aware of when working with audio is the bit rate of the digital audio signal measured in bps (bits per second).

1.3 c)

Given each sample having 16 bit resolution, what is the bit rate for the audio recording and what is the total size of the file (assuming no compression)?

[ ]:
```
### BEGIN SOLUTION
print(f"Bit rate: {fs*16} bps")
print(f"Total size of file in bits: {len(sampleData)*16}")
### END SOLUTION
```

It is often desirable to provide a graphical plot of the audio signal. Simply passing the `sampleData` to the `plot()` function should acheive this, but the x-axis will show sample index `n` instead of the time `t` in seconds, which may be rather unhelpful. Based on the total number of samples in the audio clip and the sampling rate, we can adjust the plot to show signal amplitude as a function of time t in seconds by creating a new array (e.g. `t`) of equal length to our audio signal which spans the duration of our audio signal.

For example, given an audio file with N samples and sampling rate f_s, what we want is an array `t` with `N` elements linearly spaced in the time span $0 \leq t < \frac{N}{f_s}$.

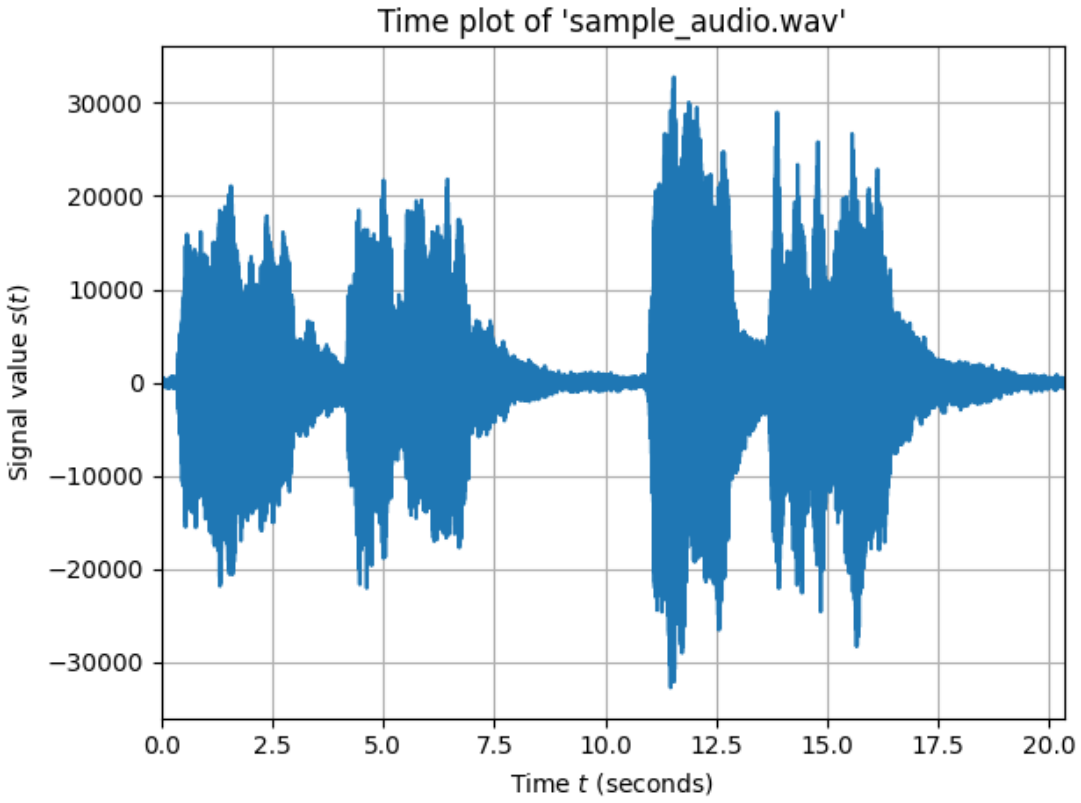
1.4 d)

Write the code to generate this plot in the codecell below. `numpy.array`, `numpy.arange` and `numpy.linspace` can all be used here. If done correctly, the resulting figure should look something like this.

[2]:
```
### BEGIN SOLUTION
import numpy as np
import matplotlib.pyplot as plt
%matplotlib ipympl

N = len(sampleData) # Sound file length in samples
t = np.linspace(0, N/fs, N, endpoint=False)

plt.plot(t, sampleData)
plt.grid(True)
plt.xlabel("Time $t$ (seconds)")
plt.ylabel("Signal value $s(t)$")
plt.title("Time plot of 'sample_audio.wav'")
plt.xlim([0, N/fs])
plt.tight_layout()
### END SOLUTION
```



2 Processing audio signals

Now that we have familiarized ourselves with the main components of an audio signal, it is time to explore some simple audio manipulation. Once again, we start by loading the contents of the audio file `sample_music.wav` using scipy.

[1]:
```
import scipy.io.wavfile as wavfile # Import module for handling of .wav audio␣
↪files
from IPython.display import Audio   # For loading embedded audio player
import numpy as np

fs, sampleData = wavfile.read("sample_audio.wav") # "fs" is sampling frequency,␣
↪"sampleData" is the sequence of measurements
print("Data type in raw sample data: ", type(sampleData[0]))
```

```
Data type in raw sample data:  <class 'numpy.int16'>
```

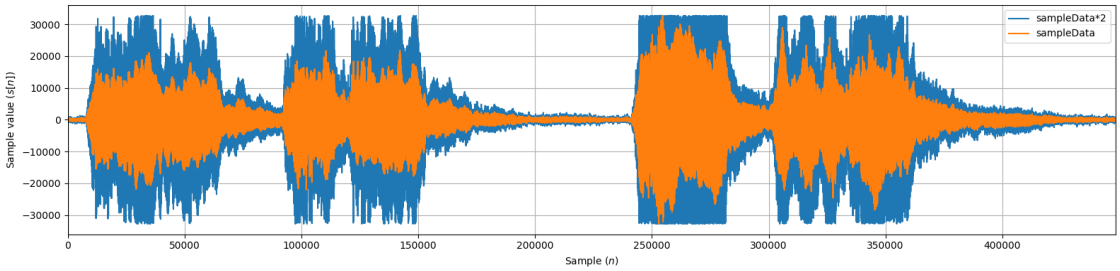
Processing audio typically involves subjecting the signal samples to som mathematical operation. However, it is important to note that each value in the `sampleData` array is of type `int16`, which is the standard sample resolution for audio recordings. When attempting to manipulate audio smaples this may result in some severe pitfalls which can be demonstrated when attempting to

scale up the values in `sampleData`.

2.1 a)

Create a figure which shows a plot of both `sampleData` and `2*sampleData`, here is a example of what this plot should look like. Give a thorough explanation of why the plot for `sampleData*2` is so evidently clipped compared to `sampleData`.

[2]:
```
### BEGIN SOLUTION
import matplotlib.pyplot as plt
plt.figure(figsize=(16,4))
plt.plot(sampleData*2, label="sampleData*2")
plt.plot(sampleData, label="sampleData")
plt.xlabel('Sample ($n$)')
plt.ylabel('Sample value ($s[n]$)')
plt.grid(True)
plt.xlim([0, len(sampleData)])
plt.legend()
plt.tight_layout()
### END SOLUTION
```



- **Answer:** What we see is the result of ***Integer Overflow***. `Int16` cannot represent values higher than 32767 or lower than −32768, so when the signal samples are multiplied by a factor of 2, this limit is in places surpassed and the sample resulting values "roll over" and start counting from the other end.

In order to avoid the issues explored in problem **a)** when processing these samples, it is usually preferable to convert the samples to floating point values (`float`). For instance, the line

```
xn = sampleData/max(abs(sampleData))
```

accomplishes this by using normal division to create a new array `xn` where the sample values are floating point values. In addition the range of values in the signal is scaled down to $-1.0 \leq x[n] \leq 1.0$, typically referred to as *normalizing* the sample values.

[3]:
```
xn = sampleData/max(abs(sampleData))
print("Data type in scaled array 'xn': ", type(xn[0]))
```

```
Data type in scaled array 'xn':  <class 'numpy.float64'>
```

We are now ready to begin adding sound effects to our music sample. One of the most useful tools available to us when modifying entire arrays is slicing, as it allows us to acces ond/or overwrite a subset of the elements in our array.

2.2 b)

Use list slicing to reduce the amplitude of the first 10 seconds of the audio recording. The resulting audio output $y(t)$ may be described in relation to the original audio signal $x(t)$ using the follwoing equation.

$$y(t) = \begin{cases} \frac{1}{8} \cdot x(t), & t \leq 10s \\ x(t), & t > 10s \end{cases}$$

[4]:
```
### BEGIN SOLUTION
yn = xn.copy()
yn[0:fs*10] *= 1/8
### END SOLUTION
```

You can use the code below to listen to the new modified audio signal in the array `yn`. Describe what you hear.

- **Answer**: A significant increase in volume is detectable around the 10 second mark.

[5]:
```
# Use the following lines to listen to the audio signal `yn`
Audio(np.int16(yn*0x7FFF), rate=fs)
```

[5]:
```
<IPython.lib.display.Audio object>
```

The audio signal in `sample_audio.wav` is what known as a *mono* signal, meaning it has only one channel. When played on a speaker system with two speakers (which is quite common), the exact same audio output will play from each of the speakers. *Stereo* audio signals are generally much more common, as having one dedicated audio stream for left and right speaker allows for more spatial depth in the listening experience. In Python, stereo signals can be represented using a $2 \times N$ 2-dimensional array. Given an audio signal consisting of the left speaker audio stream $x_{left}[n]$ and the right speaker audio stream $x_{right}[n]$, the resulting 2D array should adhere to the following pattern:

Data structure for stereo audio

Sample (n)	0	1	2	3	4	5	⋯	$N-1$
Left (row 0)	$x_{left}[0]$	$x_{left}[1]$	$x_{left}[2]$	$x_{left}[3]$	$x_{left}[4]$	$x_{left}[5]$	⋯	$x_{left}[N-1]$
Right (row 1)	$x_{right}[0]$	$x_{right}[1]$	$x_{right}[2]$	$x_{right}[3]$	$x_{right}[4]$	$x_{right}[5]$	⋯	$x_{right}[N-1]$

What we wish to do is modify our `sample_audio.wav` signal in such a way that the first 10 seconds of our audio stream sound like they originates on the left side of the listener, and the rest of the audio stream sounds like it originates to the right of the listener. This can be done by alternately reducing the amplitude of each audio channel.

Let's call our stereo audio signal $v_m(t)$, where m is channel and t i is time in seconds. Given a source single-channel audio $x(t)$ can then describe the stereo signal as follows:

$$v_m(t) = \begin{cases} x(t), & t \leq 10s \text{ and } m = 0 \\ \frac{1}{8} \cdot x(t), & t > 10s \text{ and } m = 0 \\ \frac{1}{8} \cdot x(t), & t \leq 10s \text{ and } m = 1 \\ x(t), & t > 10s \text{ and } m = 1 \end{cases}$$

2.3 c)

Create a stereo signal `v` which adheres to the specifications described above.

*P.S. Given two one-dimensional arrays **x1** and **x2** of equal length N, these can be combined into a $2 \times N$ matrix **X** using [numpy.concatenate()] as follows:*

```
X = np.concatenate(([x1], [x2]), axis=0)
```

[6]:
```
### BEGIN SOLUTION
v_left = xn.copy()
v_left[fs*10:] *= 1/8
v_right = xn.copy()
v_right[0:fs*10:] *= 1/8
vn = np.concatenate(([v_left], [v_right]), axis=0)
### END SOLUTION
```

Use the following code cell to listen to the audio signal `v`, and verify the spatial properties of the signal.

[9]:
```
# Use the following lines to listen to the audio signal `v`
Audio(np.int16(vn/np.max(np.abs(vn))*0x7FFF), rate=fs)
```

[9]:
```
<IPython.lib.display.Audio object>
```

3 Sinusoids and their frequency representation

Probably the most fundamental concept in the discipline of Signal Processing is the sinusoid (term for sine-shaped wave). We will define a siusoid mathematically with the formula

$$x(t) = A \cdot \cos\left(2\pi \cdot f \cdot t + \phi\right)$$

where A is the wave amplitude, f is the frequency and ϕ is the phase component.

The reason for the emphasis on sinusoids in signal processing is most (if not all) frequency analysis is founded on decomposing any signal into a sum of component sinusoids. When viewing the magnitude spectrum of any signal $s(t)$, the level shown in the frequency curve for a given point f along the x-axis (frequency) tells us how much of the signal can be attributed to a sinusoid with the exact frequency f.

Below is an example of python code to generate a signal $x(t)$ (`xt`) consisting solely of one sinusoid with frequency $f_1 = 1$Hz, amplitude $A_1 = 1$, and phase $\phi_1 = -\frac{\pi}{3}$. Once the signal `xt` is created, we use `matplotlib.pyplot` to present both a view of the shape of $x(t)$ in the time domain, and it's corresponding magnitude spectrum $|X(f)|$ in the frequency domain. What is important to note,

is that the magnitude spectrum of $x(t)$ is equal to zero for all frequencies *except* for $f = 1\text{Hz}$, as represented by the sudden spike.

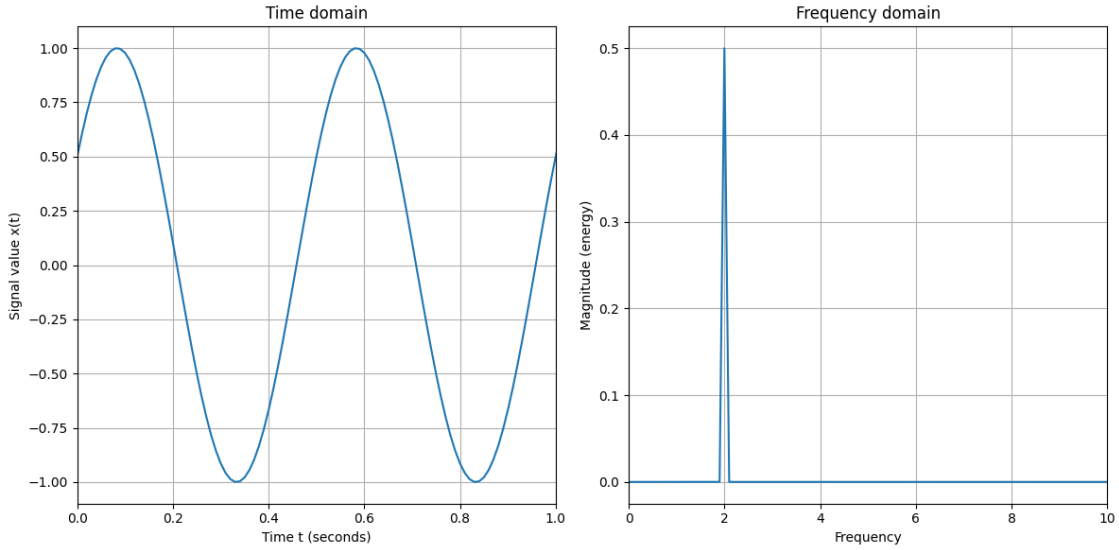
[3]:
```
import numpy as np
import matplotlib.pyplot as plt

# Generate Signal!
fs = 100  # Sampling Frequency
T = 20   # Signal Duratino'
A1 = 1   # Amplitude
F1 = 2   # Wave frequency
phi1 = -np.pi/3  # Phase
t = np.linspace(0, T, T*fs, endpoint=False) # Create array of time values
xt = A1*np.cos(2*np.pi*F1*t+phi1) # Create sinusoid signal

plt.figure(figsize=(12,6)) # Create "blank" figue of specified size

# PLot sinusoid in time domain
plt.subplot(1,2,1)
plt.plot(t, xt)
plt.xlim([0, T/20]) # Zoom in on x-axis
plt.xlabel("Time t (seconds)")
plt.ylabel("Signal value x(t)")
plt.title("Time domain")
plt.grid(True)
plt.subplot(1,2,2)

# Plot sinusoid in frequency domain
plt.magnitude_spectrum(xt, Fs=fs) # argument 'fs' required for x-axis to␣
 ↪represent Hertz(Hz)
plt.xlim([0, fs/10]) # Zoom in on x-axis
plt.title("Frequency domain")
plt.grid(True)
plt.tight_layout() # Make room for axis labels
```



3.1 a)

Using the above example as a template, add two new frequency components to the signal $x(t)$. Their properties are as follows:

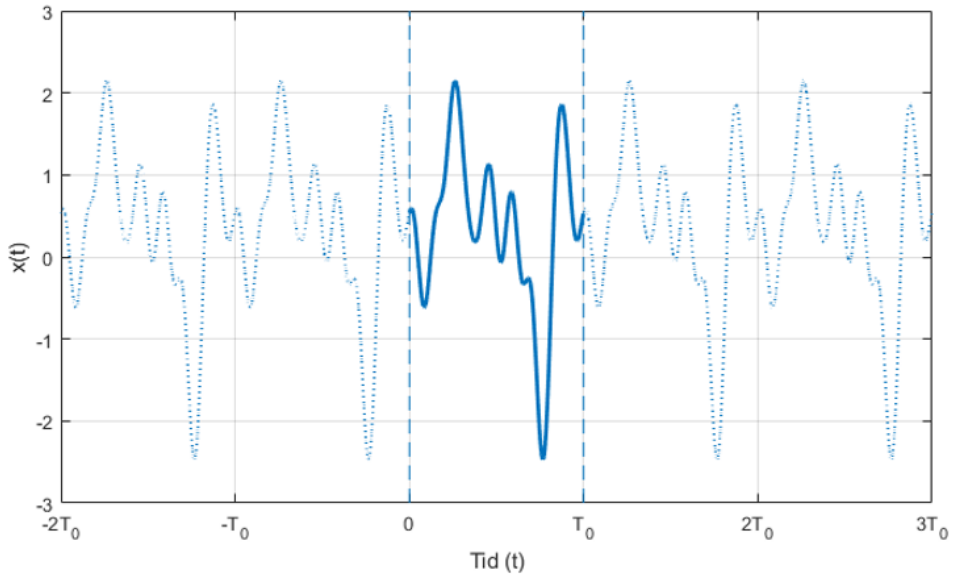
Frequency f (Hz)	Amplitude A	Phase ϕ (radians)
6	$\frac{1}{3}$	0
10	$\frac{1}{5}$	$\frac{\pi}{3}$

3.2 b)

Study the height of the spikes in the magnitude spectrum. Do they match the amplitudes used when generating the signal? If not, can you find any consistency in the relation between spike height and sinusoid component amplitudes?

- They do not match. Spike height accounts for $\frac{1}{2}$ of amplitude.

For any periodic signal, the ***fundamental frequency*** $f_0 = \frac{1}{T_0}$ is defined as the number of times a given signal pattern repeats itself *exactly* during the ocurse of one second. An intuitive approach to measuring the fundamental frequency is measuring the *fundamental period* T_0 (i.e. the distance along the time axis until a wave pattern repeats) as shown in the figure below.



The mathematical definition of the fundamental period / fundamental frequncy can be given as:

$$x(t) = x\left(t - \frac{k}{f_0}\right) = x\left(t - k \cdot T_0\right), \quad k \in \mathbb{Z}$$

where k is any given integer value.

While signals in encountered in the real world seldom are 100% periodic, audio signals especially tend to have a very significant periodic component which usually determines the pitch of the tone heard.

3.3 c)

Study the time series plot of the signalfrom task **a)**. What is the fundamental period T_0?

- $T_0 = 1\text{s}$

The fundamental frequency f_0 of a signal can also be determined by studying the magnitude spectrum. Here, the task is to find the largest possible value for f_0 where *all* the significant spikes along the frequency axis occur at frequencies which are a multiple of f_0. That is, $f \in k \cdot f_0, \quad k = \{0, 1, 2, ...\}$.

3.4 d)

By use of the method explaied above, what fundamental frequency can we determine from the frequency plot? Does it match our answer from task **c)**?

- $f_0 = 1\text{Hz} = \frac{1}{1\text{s}}$

4 Graphical analysis of an audio signal

It is time to use Python to conduct some more in-depth graphical analysis of our audio signal `sample_audio.wav`. Start by running the code cell below which loads the contents of the audio file, and creates a scaled array `x` with sample values as floating-point numbers.

[ ]:
```
import scipy.io.wavfile as wavfile # Import module for handling of .wav audio␣
 ↪files
from IPython.display import Audio  # Import the Audio object for use in audio␣
 ↪playback
import numpy as np

fs, sampleData = wavfile.read("sample_audio.wav") # "fs" is sampling frequency,␣
 ↪"sampleData" is the sequence of measurements
x = sampleData/max(abs(sampleData))               # Scale sample values to the␣
 ↪range -1 < x[n] < 1
```

We now wish to analyze a short extract of the audio file to see if we can identif the fundamental frequency f_0 at around $t = 12.3$ seconds.

4.1 a)

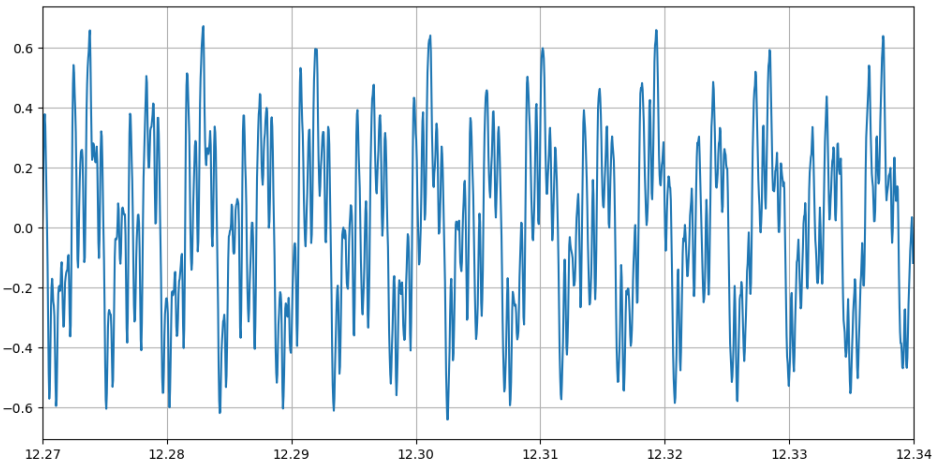
Create a figure showing the signal $x(t)$ during the time interval $12.27s \leq t \leq 12.33s$. Use the same method as in task 3 to measure the fundamental period from the time plot. The plot should look something like this.

Tip 1: In order to assure higher precision, measure the time over a larger number of repetitions (e.g. 5), and use this to calculate the average period

Tip 2: The cell command ***`%matplotlib ipympl`*** *enables interactive plots. You can now zoom in, or use the mouse cursor to get a relatively accurate reading of a point in the figure.The downside to this is figues remaining "active" across code cells. Use the function* ***`figure()`*** *to create a new blank figure whenever you want to make a new plot.*

[2]:
```
import matplotlib.pyplot as plt
%matplotlib ipympl
### BEGIN SOLUTION
t1 = 12.27 # seconds
t2 = 12.34 # seconds
n1 = round(fs*t1)
n2 = round(fs*t2)
plt.close(1); plt.figure(1, figsize=(12,6))
plt.plot(np.linspace(t1, t2, n2-n1, endpoint=False), x[n1:n2])
plt.grid(True)
plt.xlim([t1, t2])
### END SOLUTION
```

[2]:
```
(12.27, 12.34)
```



1. Time for 5 repetitions: $\Delta t = 12.3375 - 12.292 = 0.0455$
2. Fundamental period $T_0 = \frac{0.0365}{5} = 0.0091$
3. Fundamental frequency $f_0 = \frac{1}{T_0} = \frac{1}{0.0091} = 109.89$

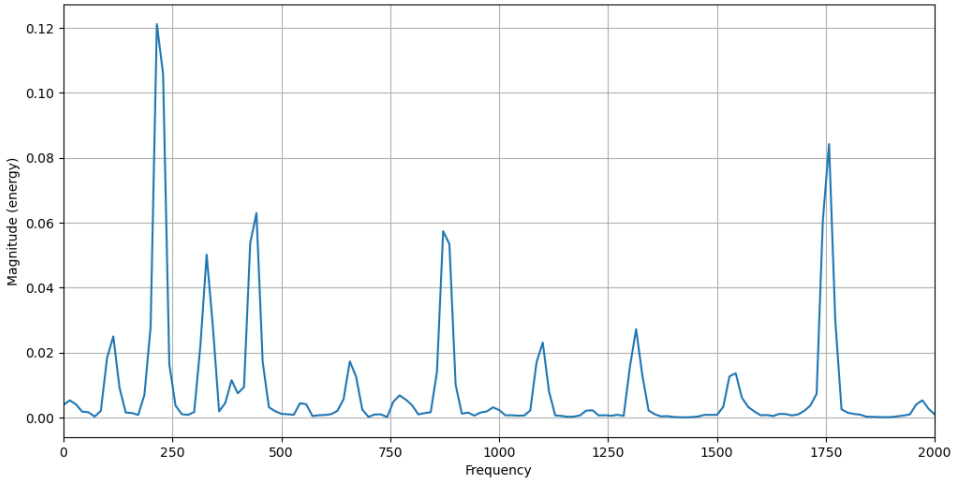
Measuring the fundamental frequency based on the signal curve $x(t)$ can often be quite difficult, as it depends on us correctly identifying a repeating pattern. It is usually preferred to analyze the magnitude spectrum $|X(f)|$ instead, where a periodic signal manifests as a series of uniformly spaced "spikes".

4.2 b)

Use the function `matplotlib.pyplot.magnitude_spectrum()` as introduced in task 3 to show the magnitude spectrum $|X(f)|$ of $x(t)$ for the same time interval $12.27s \leq t \leq 12.33s$. Measure the space between spikes to find the fundamental frequency. *P.S. If we zoom in on the lower end of the frequency axis ($0Hz \leq f \leq 2000Hz$), the magnitude spectrum $|X(f)|$ should look something like this.*

[3]:
```
import matplotlib.pyplot as plt
%matplotlib ipympl
### BEGIN SOLUTION
t1 = 12.27 # seconds
t2 = 12.34 # seconds
n1 = round(fs*t1)
n2 = round(fs*t2)
plt.close(2); plt.figure(2, figsize=(12,6))
plt.magnitude_spectrum(x[n1:n2], fs)
plt.grid(True)
plt.xlim([0, 2000])
### END SOLUTION
```

[3]: (0.0, 2000.0)



- Detect four first spikes: $f \in \{112, 213, 325, 445\}$
- Spaces: $112 - 0 = 112$, $213 - 112 = 101$, $325 - 213 = 212$, $445 - 325 = 120$
- Average space: $\frac{112+101+212+120}{4} = 111.25Hz = f_0$
- Very close to our answer in **a)**

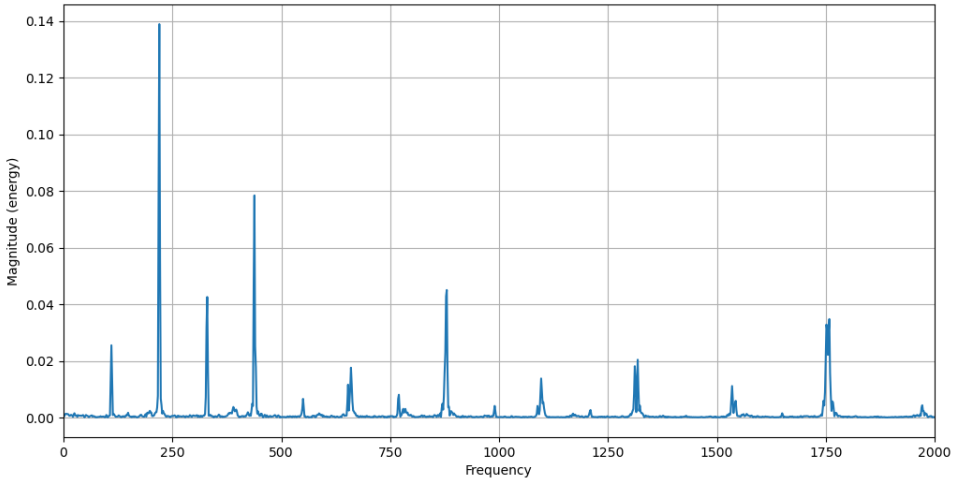
Our magnitude spectrum plot in task **b)**, while providing some good information appears to be quite jagged. This is because our *resolution* along the frequency axis depends on the lenght of the signal interval subject to frequency analysis. To get a more detailed magnitude spectrum, we can increase the analysis interval.

4.3 c)

Create another plot of the magnitude spectrum $|X(f)|$, this time based on the values of $x(t)$ for $12.0\text{s} \leq t < 12.6\text{s}$. How does the increased resoution affect the magnitude spectrum plot?

[4]:
```
import matplotlib.pyplot as plt
%matplotlib ipympl
### BEGIN SOLUTION
t1 = 12.0 # seconds
t2 = 12.6 # seconds
n1 = round(fs*t1)
n2 = round(fs*t2)
plt.close(3); plt.figure(3, figsize=(12,6))
plt.magnitude_spectrum(x[n1:n2], fs)
plt.grid(True)
plt.xlim([0, 2000])
### END SOLUTION
```

[4]: (0.0, 2000.0)



- The spikes become much thinner, as we have much closer neighboring data points with low frequency content.

Generally, higher frequency resolution is helpful when conducting frequency analysis. However, we also run the risk of "smearing" the frequency plot *if* the tone subject to analysis were to change during our analysis window. We also have no way of knowing whether the tone lasted the entire 0.6 seconds we analyzed, or only parts of it. In other words, there is always a tradeoff between frequency resolution and accuracy in the time domain. This is one manifestation of what is referred to as the uncertainty principle, which is explained beautifully in this youtube-video.

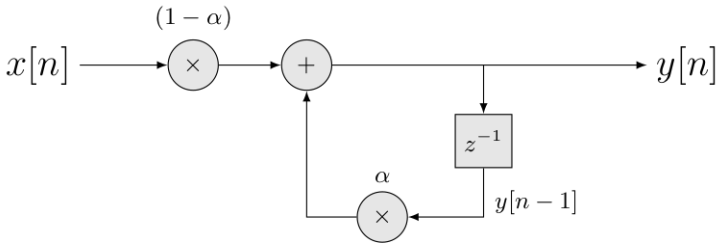
5 Simple Audio Filtering

Digital filters are a major part of many digital signal processing systems. While there are a lot of advanced topics in regards to digital filters which we will explore later in the course, making use of a simple lowpass filter can be done quite easily.

All linear and time-invariant (LTI) digital systems will be comprised of three main operations:

- Multiplication of a signal sample $x[n]$ by some constant term.
- Adding together two signal samples $x_1[n]$ and $x_2[n]$.
- Delaying a sample by N sample periods (represented as z^{-N}).

We can combine these three operations to create an exponential averaging filter as shown in the system model illustrated below.



Such a system may be summarized mathematically as a numerical algorithm for calculating each output sample $y[n]$:

$$y[n] = (1-\alpha) \cdot x[n] + \alpha \cdot y[n-1] \tag{5.1}$$

Translated into python code, each output value `current_y` will be calculated by combining the current input value `current_x`, and the previous output value `previous_y` with individual weights determined by a filter coefficient `alpha`.

```
current_y = (1-alpha)*current_x + alpha*previous_y
previous_y = current_y
```

The lines of code above can process *one sample*, and store the current output in the variable `previous_y`, so it can be used when filtering the next input sample. To filter an entire audio signal, we must iterate through the entire audio signal array, doing this operation with each sample in turn.

[1]:
```
import scipy.io.wavfile as wavfile # Import module for handling of .wav audio␣
↪files
from IPython.display import Audio  # Import the Audio object for use in audio␣
↪playback
import numpy as np

fs, sampleData = wavfile.read("sample_audio.wav") # "fs" is sampling frequency,␣
↪"sampleData" is the sequence of measurements
xn = sampleData/max(abs(sampleData))                # Scale sample values to the␣
↪range -1 < x[n] < 1
```

5.1 a)

Use the filter in equation 5.1 with $\alpha = 0.95$ to filter the audio signal from `sample_audio.wav`, and play the sound using your computer. What do you hear?

[3]:
```
### BEGIN SOLUTION
alpha = 0.95
yn = np.zeros(len(xn))
for i, x in enumerate(xn):
    yn[i] = yn[i-1]*alpha + (1-alpha)*xn[i]
Audio(yn, rate=fs)
### END SOLUTION
```

[3]:
```
<IPython.lib.display.Audio object>
```

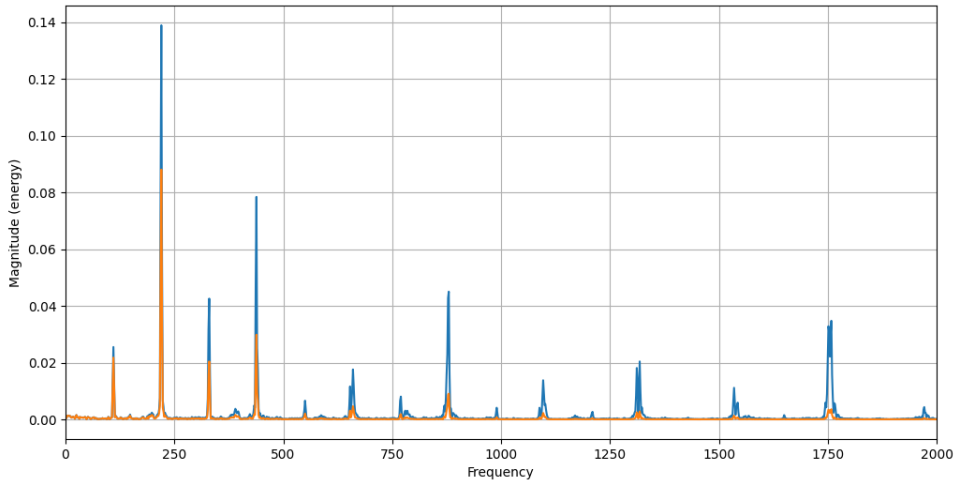
We hear a "muffled" version of the original audio recording, where low frequency elements have a much greater relative emphasis compared to the original audio file

5.2 b)

Create a plot showing the frequency content of the audio signal around $t = 12.3$ s before and after filtering. What is the difference between the two plots, and how do our plots reflect the filtering effects we heard in task **a)**?

[12]:
```
import matplotlib.pyplot as plt
%matplotlib ipympl
### BEGIN SOLUTION
t1 = 12.0 # seconds
t2 = 12.6 # seconds
n1 = round(fs*t1)
n2 = round(fs*t2)
plt.close(3); plt.figure(3, figsize=(12,6))
plt.magnitude_spectrum(xn[n1:n2], fs, label="Original")
plt.magnitude_spectrum(yn[n1:n2], fs)
plt.grid(True)
plt.xlim([0, 2000])
### END SOLUTION
```

[12]:
```
(0.0, 2000.0)
```



As seen in the frequency plot, the first harmonic component at ≈ 110 Hz as a magnitude roughly quivalent to the original. From here, the 1st harmonic is attenuated a little more, the 2nd harmonic

is attenuated by roughly halv, and the attenuation continues to increase for harmonics 3, 4 and so on. This is quite typical of a **lowpass filter**.

6 Review notes for exercise 1

Exercises are approved in-person during lab session for the course. You will be asked to show your work to a lecturer or student assistant, and the two of you will go through the assignment together. In addition to being a mechanism for approving assignments, these conversations are a good opportunity to clear up any lingering questions or concepts which you're not 100% sure about.

6.1 Notes for weekly review

You are encouraged to write a few bullet points on your experience with the exercise, to be used as a guiding reference to the conversation with the student assistant. Here are a few questions to get you started.

- Were you able to complete all questions in the exercise?
- What part of the exercise did you find challenging?
- What was positive about this week's exercise?
- What part of the exercise could have been better designed?
- Is there anything in particular you wish to ask about?

6.2 End-of-year review

Before handing in the exercise reports as part of your portfolio hand-in, take a few minutes to skim through the assignment and look through the notes you wrote for the weekly review and answer the following questions:

- If there were particularly challenging topics in this assignment, have the learning activities since then helped you understand said topics and if so, how?
- Were there any particular parts of this assignment which proved particularly relevant when working with the portfolio project?