COM3502-4502-6502 Speech Processing - Python Programming Assignment

1. General Information

This programming assignment is worth 55% of the overall course mark.

You are free to complete this assignment in your own time. However, feedback, advice and guidance is available during the lab classes and via the discussion board on Blackboard.

Note: Via these channels we try to help you as much as possible, but will not debug your code or provide solutions to the assignment itself.

Note: It will take some time to complete this assignment, so plan your work accordingly over the coming weeks. Read these instructions carefully.

Note: Please be aware that students registered on COM4502 and COM6502 have **additional tasks** to perform. These are marked 'COM4502-6502 Only'.

Note: You should always ensure that your results (e.g. in terms of plots you create) are clear to undrestand and leave no room for mis-interpretation. This can often easily achieved by adding proper x- and y-axis labels, titles, legends etc. Where results are not clear to interprete, this might result in missed points.

1.1 Student Data

Student Family Name: **Amoiridou**

Student Given Name(s): **Evangelia**

Date of submission: **Friday 16th of December 2022**

Academic Year 2021/2022

1.2 Copyright

1.3 Hand-In Procedure and Deadline

Once you have completed the assignment you should submit a `.zip` file (via Blackboard) containing your solution (as a file named `YourName.ipynb`) and possibly other source linked in your Jupyter Notebook. Also the `.zip` filename should be of the form `YourName.zip` . Please make also sure that your name is entered correctly in the section above.

Standard departmental penalties apply for late hand-in (https://sites.google.com/sheffield.ac.uk/comughandbook/general-information/assessment/late-submission) and plagiarism (https://sites.google.com/sheffield.ac.uk/comughandbook/general-information/assessment/unfair-means).

The **deadline** for handing-in this assignment (via Blackboard) is **Friday 16th December 2022**

Task 0:

Ensure that you data is correctly entered in the section at the top of this sheet and that the filename is in the form `AmoiridouEvangelia.ipynb` .

1.4 Libraries

You should be familar to the use of the following Python libraries from the lab. You should not need to use additional ones. You are allowed to use additional libraries if necessary for your code. If they need to be installed by `!pip install <libraryname>` or `!conda install <libraryname>` , please indicate this as a comment in your code. You should not make use of libraries that can't be installed by either `!pip install` or `!conda install` . You have to ensure that your Notebook runs "out of the box". You can test this on the Computer Lab machines in the Diamond if you are unsure and using your own computer.

In [ ]:

```
#Let's do some necessary and nice-to-have imports
%matplotlib inline
import matplotlib.pyplot as plt    # plotting
#import seaborn as sns; sns.set()  # styling
import numpy as np                 # math

import soundfile as sf             # to load files
from IPython import display as ipd # for sound playback

from scipy import signal           # filter designs (if not already imported)
```

2. Download, load, and analyse audio

Task T1:

- Load a wave file containing speech. You can find a file at https://staffwww.dcs.shef.ac.uk/people/S.Goetze/sound/speech.wav (https://staffwww.dcs.shef.ac.uk/people/S.Goetze/sound/speech.wav) and should be able to download this. You can also use your own WAVE files if you prefer this. If you want to record WAVE files and are using your own computer, the program Audacity (https://www.audacityteam.org/download/) is one possibility to record WAVE files (https://manual.audacityteam.org/man/basic_recording_editing_and_exporting.html).
- Visualise the signal in time domain, in the spectral domain (as spectrum) and as a time-frequency representation (spectrogram). Please ensure proper axis labels for all your plot in this assignment.
- Playback the signal.

In [ ]:

```
# your code here

# download file containing speech
!curl https://staffwww.dcs.shef.ac.uk/people/S.Goetze/sound/speech.wav -o speech.wav

import soundfile as sf
s_file_name = 'speech.wav'
s,fs = sf.read(s_file_name)


# visualise signal in time domain
plt.figure(figsize=(9,17))
t=np.linspace(0,len(s)/fs,len(s)) #create time vector
plt.subplot(3,1,1)
plt.plot(t,s)
plt.title('Visualisation of the signal in time domain')
plt.xlabel('$t$ in seconds');
plt.ylabel('$s(t)$');

# visualise signal in spetral domain
# calculate the spectum (frequency domain representation)
FFT_length = 2**15 # take a power of two which is larger than the signal length
f = np.linspace(0, fs/2, num=int(FFT_length/2+1))
spectrum = np.abs(np.fft.rfft(s,n=FFT_length))
print(spectrum.shape)
print(f.shape)

# plot the spectrum
plt.subplot(3,1,2)
plt.plot(f,spectrum)
plt.title('Visualisation of the signal in spectal domain')
plt.xlabel('frequency $f$ in Hz')
plt.ylabel('$x(f)$')

# visualise signal as spectogram
plt.subplot(3,1,3)
plt.specgram(s, Fs=fs);
plt.title('Spectrogram of the signal')
plt.xlabel('time $t$')
plt.ylabel('frequency $f$')

plt.colorbar(label='dB');
plt.clim(-150,0)

# playback the signal
import IPython.display as ipd
ipd.Audio(s/10,rate=fs)
```

Question Q1:

- Determine the sampling frequency f_s in Hz and the length of the signal in seconds. What is the sampling interval T_s of your signal? What is the highest occuring frequency?

Note: You can either give your answer in form of a code block (e.g. by using the `print()` functions) or as text. For the latter, change the type of the next cell (https://jupyter-notebook.readthedocs.io/en/stable/notebook.html#structure-of-a-notebook-document) from `code` to `markdown` or use the yellow example text below.

In [ ]:

```
# Your answer here

# print sampling frequency of the signal
print ("The sampling frequency of the signal is "+str(fs)+" Hz.")
# print length of the signal
print ("The length of the signal is "+str(len(s)/fs)+" sec.")
# print samplig interval
print ("The sampling inerval of the signal is "+str(1/fs)+" sec.")
# print highest occuring frequency
print ("The highest occuring frequency of the signal is approximately 6500 Hz because it's the highest frequency in which energ
```

In case you want to answer by written text, we would appreciate if you colour-code your answers, e.g. like using orange font colour as illustrated in this example. This helps us, not to overlook parts of your answers.

3. Piece-wise linear filtering in the time domain

Task T2:

- Design a high-pass filter with a cut-off frequency of ≈ 500 Hz using a filter design method of your choice.
- Design a low-pass filter with a cut-off frequency of ≈ 500 Hz.
- Design a band-stop filter with a cut-off frequencies of ≈ 300 Hz and of ≈ 1.1 kHz.
- Simulate the effect of a land-line telephone by eliminating all energy below 300 Hz and above $3,400$ Hz.
- Visualise the transfer functions of the filters and the zero-pole plots.
- Apply the designed filters, compare filter input and output as a time-frequency visualisation and play back the filtered signal.

Note: Don't forget proper labeling / description of your figures to make clear what is what.

Note: In case you encounter stability problems, rememer that we mentioned in the lecture, that filters can be designed as second-oder-systems (SOS) which the design methods you are familia with can realise.

In [ ]:

```
# Your code here

# high pass filter tolerance function
def plot_tolerance_scheme_hp(Wp=0.25,Ws=0.3,Rp_lin=0.9,Rs_lin=0.1):
    dh1x=[Ws,1];  dh1y=[1,1];            # (x,y) coordinates of lines
    dh2x=[Wp,1];  dh2y=[Rp_lin,Rp_lin];
    dv2x=[Wp,Wp]; dv2y=[0,Rp_lin];
    sh1x=[0,Ws];  sh1y=[Rs_lin,Rs_lin];
    sh2x=[0,Wp];  sh2y=[0,0];
    svx=[Ws,Ws];  svy=[Rs_lin,1];

    # plot the actual lines
    plt.plot(dh1x,dh1y,'k--',dh2x,dh2y,'k--',dv2x,dv2y,'k--',sh1x,sh1y,'k--',
             sh2x,sh2y,'k--',svx,svy,'k--');
    plt.xlabel('Frequency $\Omega/\pi$');
```

In [ ]:

```
# design high-pass filter, cut-off frequency 500Hz using butterworth method
# assuming 10% tolerance, fs=8000Hz ripples=10%

from scipy import signal as sig         # filter designs

wp1=0.025;                  # passband edge frequency
ws1=0.020;                  # stopband edge frequency
Rp_lin1=0.8;                # allowed ripples in the pass band area
Rs_lin1=0.2;                # allowed ripples in the stop band area

Rp1=-20*np.log10(Rp_lin1);  # max passband ripple in dB
Rs1=-20*np.log10(Rs_lin1);  # min stopband attenuation in dB
plot_tolerance_scheme_hp(wp1,ws1,Rp_lin1,Rs_lin1)

# get lowest filter order N to fullfill requirements above
N1, Wn1 = sig.buttord(wp1, ws1, Rp1, Rs1)
b1, a1 = sig.butter(N1, Wn1, 'high')

print('The minimum possible filter order to fulfil the tolerance scheme is '+str(N1)+'.')
print('The cut-off frequency which will be {:.2f}.'.format(Wn1))

b1, a1 = signal.butter(N1, Wn1, 'highpass')
f1,h1 = signal.freqz(b1,a1)
omega1 = np.linspace(0,1,len(f1))

plot_tolerance_scheme_hp(wp1,ws1,Rp_lin1,Rs_lin1)
plt.plot([Wn1,Wn1],[0,1],color='red',ls=':',label='cutoff frequency')
plt.plot(omega1,np.abs(h1), lw=2, label='Butterworth high-pass')

plt.title('Butterworth h-pass filter of order ' + str(N1))
plt.ylabel('Amplitude $|h(e^{j \Omega})|$')
plt.legend();


# visualise transfer functions and zero-pole pots of the filters

# plot frequency response of high-pass filter, cut-off frequency 500Hz
fig = plt.figure(figsize=(8,4)) # create a figure of size 8 x 4 inches
h1 = np.abs(np.fft.fft(b1,1024))/np.abs(np.fft.fft(a1,1024));
h1 = h1[0:513] # only show first half (positive frequencies)
omega = np.linspace(0,1,513)

plt.subplot(1,2,1)
plt.plot(omega, abs(h1), lw=2)
plt.title('Butterworth high-pass filter \n f>500Hz')
plt.ylabel('Amplitude $|h(e^{j \Omega})|$');

def zplane(z, p, title='Poles and Zeros'):
    "Plots zeros and poles in the complex z-plane"
    ax = plt.gca()

    ax.plot(np.real(z), np.imag(z), 'bo', fillstyle='none', ms=10)
    ax.plot(np.real(p), np.imag(p), 'rx', fillstyle='none', ms=10)
    unit_circle = plt.Circle((0, 0), radius=1, fill=False,
                             color='black', ls='--', alpha=0.9)
    ax.add_patch(unit_circle)

    plt.title(title)
    plt.xlabel('Re{$z$}')
    plt.ylabel('Im{$z$}')
    plt.axis('equal')

# plot zero-pole plot of high-pass filter, cut-off frequency 500Hz
plt.subplot(1,2,2)
zplane(-1, np.roots(a1))
plt.text(-0.95,0.1,'('+str(N1)+')');
```

In [ ]:

```
# low pass filter tolerance function

def plot_tolerance_scheme(Wp=0.020,Ws=0.025,Rp_lin=0.8,Rs_lin=0.2):
    dh1x=[0,Ws];  dh1y=[1,1];            # (x,y) coordinates of lines
    dh2x=[0,Wp];  dh2y=[Rp_lin,Rp_lin];
    dv2x=[Wp,Wp]; dv2y=[0,Rp_lin];
    sh1x=[Ws,1];  sh1y=[Rs_lin,Rs_lin];
    sh2x=[Wp,1];  sh2y=[0,0];
    svx=[Ws,Ws];  svy=[Rs_lin,1];
    # plot the actual lines
    plt.plot(dh1x,dh1y,'k--',dh2x,dh2y,'k--',dv2x,dv2y,'k--',sh1x,sh1y,'k--',
             sh2x,sh2y,'k--',svx,svy,'k--');
    plt.xlabel('Frequency $\Omega/\pi$');
```

In [ ]:

```
# design low-pass filter, cut-off frequency 500Hz using butterworth method
# assuming 10% tolerance, fs=8000Hz ripples=10%

wp2=0.020;                  # passband edge frequency
ws2=0.025;                  # stopband edge frequency
Rp_lin2=0.8;                # allowed ripples in the pass band area
Rs_lin2=0.2;                # allowed ripples in the stop band area

Rp2=-20*np.log10(Rp_lin2);  # max passband ripple in dB
Rs2=-20*np.log10(Rs_lin2);  # min stopband attenuation in dB
plot_tolerance_scheme_hp(wp2,ws2,Rp_lin2,Rs_lin2)

# get lowest filter order N to fullfill requirements above
N2, Wn2 = sig.buttord(wp2, ws2, Rp2, Rs2)
b2, a2 = sig.butter(N2, Wn2, 'low')

print('The minimum possible filter order to fulfil the tolerance scheme is '+str(N2)+'.')
print('The cut-off frequency which will be {:.2f}.'.format(Wn2)) # format number - two digits after decimal pt.


# visualise transfer functions and zero-pole pots of the filters

# plot frequency response of low-pass filter, cut-off frequency 500Hz
fig=plt.figure(figsize=(8,4))         # create a figure of size 8 x 4 inches
h2=np.abs(np.fft.fft(b2,1024))/np.abs(np.fft.fft(a2,1024));
h2=h2[0:513]                          # only show first half (positive frequencies)
omega=np.linspace(0,1,513)

plt.subplot(1,2,1)
plt.plot(omega, abs(h2), lw=2)
plt.title('Butterworth low-pass filter \n f<500Hz')
plt.ylabel('Amplitude $|h(e^{j \Omega})|$');

def zplane(z, p, title='Poles and Zeros'):
    "Plots zeros and poles in the complex z-plane"
    ax = plt.gca()

    ax.plot(np.real(z), np.imag(z), 'bo', fillstyle='none', ms=10)
    ax.plot(np.real(p), np.imag(p), 'rx', fillstyle='none', ms=10)
    unit_circle = plt.Circle((0, 0), radius=1, fill=False,
                             color='black', ls='--', alpha=0.9)
    ax.add_patch(unit_circle)

    plt.title(title)
    plt.xlabel('Re{$z$}')
    plt.ylabel('Im{$z$}')
    plt.axis('equal')

# plot zero-pole plot of lowh-pass filter, cut-off frequency 500Hz
plt.subplot(1,2,2)
zplane(-1, np.roots(a2))
plt.text(-0.95,0.1,'('+str(N2)+')');
```

In [ ]:

```
# design band-stop filter, cut-off frequency 300Hz and 1100HZ using butterworth method

wp3 = [0.016, 0.04]       # pass-band frequency limits (normalised)
ws3 = [0.01, 0.06]        # stop-band frequency limits (normalised)
Rp3 = 1                   # allowed ripples in the pass band area (1 dB)
Rs3 = 20                  # allowed ripples in the stop band area (20dB)

# determine necessary filter order as well as cut-off frequencies
N3, Wn3 = signal.buttord(wp3, ws3, Rp3, Rs3)
b3, a3 = sig.butter(N3, Wn3, 'bandstop')
f3,h3 = sig.freqz(b3,a3)
omega3 = np.linspace(0,1,len(f3))
plt.plot(omega3, np.abs(h3))

print('The minimum possible filter order to fulfil the tolerance scheme is '+str(N3)+'.')
print('The 1st cut-off frequency which will be {:.2f}.'.format(Wn3[0]))
print('The 2nd cut-off frequency which will be {:.2f}.'.format(Wn3[1]))

# plot frequency response of band-stop filter, cut-off frequencies 300Hz and 1100Hz
f3,h3 = sig.freqz(b3,a3)
omega3 = np.linspace(0,1,len(f3))
fig = plt.figure(figsize=(8,4))      # create a figure of size 8 x 4 inches

plt.subplot(1,2,1)
plt.plot(omega3, np.abs(h3))
plt.title('Butterworth band-stop filter \n 300Hz>f and  f>1100Hz')
plt.ylabel('Amplitude $|h(e^{j \Omega})|$')
plt.xlabel('Frequency $\Omega / \pi$')
plt.grid(True,which='both', axis='both')

# plot zero-pole plot of band-stop filter, cut-off frequencies 300Hz and 1100Hz
plt.subplot(1,2,2)
zplane(-1, np.roots(a3))
plt.text(-0.95,0.1,'('+str(N3)+')');
```

In [ ]:

```
#design band-pass filter, cut-off frequency 300Hz and 3400HZ using butterworth method
wp4 = [0.018,0.10]       # pass-band frequency limits (normalised)
ws4 = [0.008,0.2]       # stop-band frequency limits (normalised)
Rp4 = 1                  # allowed ripples in the pass band area (1 dB)
Rs4 = 20                 # allowed ripples in the stop band area (20dB)
Rp_lin4 = 10**(-Rs4/20); # transforming dB back to linear
Rs_lin4 = 10**(-Rs4/20)

# determine necessary filter order as well as cut-off frequencies
N4, Wn4 = signal.buttord(wp4, ws4, Rp4, Rs4)
b4, a4 = sig.butter(N4, Wn4, 'bandpass')
f4,h4 = sig.freqz(b4,a4)
omega4 = np.linspace(0,1,len(f4))
plt.plot(omega4, np.abs(h4))

print('The minimum possible filter order to fulfil the tolerance scheme is '+str(N4)+'.')
print('The 1st cut-off frequency which will be {:.2f}.'.format(Wn4[0]))
print('The 2nd cut-off frequency which will be {:.2f}.'.format(Wn4[1]))

# plot frequency response of band-stop filter, cut-off frequencies 300Hz and 3400Hz
f4,h4 = sig.freqz(b4,a4)
omega4 = np.linspace(0,1,len(f4))
fig = plt.figure(figsize=(8,4))

plt.subplot(1,2,1)
plt.plot(omega4, np.abs(h4))
plt.title('Butterworth band-pass filter \n 300Hz< f <3400Hz')
plt.ylabel('Amplitude $|h(e^{j \Omega})|$')
plt.xlabel('Frequency $\Omega / \pi$')
plt.grid(True,which='both', axis='both')

# plot zero-pole plot of band-stop filter, cut-off frequencies 300Hz and 3400Hz
plt.subplot(1,2,2)
zplane(-1, np.roots(a4))
plt.text(-0.95,0.1,'('+str(N4)+')');
```

In [ ]:

```
s_filtered1 = signal.filtfilt(b1, a1, s)   #high pass
s_filtered2 = signal.filtfilt(b2, a2, s)   #low pass
s_filtered3 = signal.filtfilt(b3, a3, s)   #bandstop
s_filtered4 = signal.filtfilt(b4, a4, s)   #bandpass

ipd.Audio(s_filtered4, rate=fs)
```

In [ ]:

```
# visualise as spectrograms
fig=plt.figure(figsize=(15,6))    # create a figure of size 15 x 6 inches

plt.subplot(1,5,1)
plt.specgram(s, Fs=fs)
plt.title('spectrogram of signal \n before filtering')
plt.xlabel('time $t$')
plt.ylabel('frequency $f$')
plt.grid(False)                   # no grid (in case you used seaborn)

plt.colorbar(label='dB');
plt.clim(-180,-30)                # select the 'color'/amplitude range (in dB)

plt.subplot(1,5,2)
plt.specgram(s_filtered1, Fs=fs)
plt.title('spectrogram of filtered signal \n high-pass f>500')
plt.xlabel('time $t$')
plt.ylabel('frequency $f$')
plt.grid(False)                   # no grid (in case you used seaborn)

plt.colorbar(label='dB');
plt.clim(-180,-30)

plt.subplot(1,5,3)
plt.specgram(s_filtered2, Fs=fs)
plt.title('spectrogram of filtered signal \n low-pass f<500')
plt.xlabel('time $t$')
plt.ylabel('frequency $f$')
plt.grid(False)                   # no grid (in case you used seaborn)

plt.colorbar(label='dB');
plt.clim(-180,-30)
plt.subplot(1,5,4)
plt.specgram(s_filtered3, Fs=fs)
plt.title('spectrogram of filtered signal \n 300Hz>f and f>1100Hz')
plt.xlabel('time $t$')
plt.ylabel('frequency $f$')
plt.grid(False)                   # no grid (in case you used seaborn)

plt.colorbar(label='dB');
plt.clim(-180,-30)

plt.subplot(1,5,5)
plt.specgram(s_filtered4, Fs=fs)
plt.title('spectrogram of filtered signal \n 300Hz< f <3400Hz')
plt.xlabel('time $t$')
plt.ylabel('frequency $f$')
plt.grid(False)                    # no grid (in case you used seaborn)

plt.colorbar(label='dB');
plt.clim(-180,-30)

plt.tight_layout()                 # to see all axis descriptions

#play back the filtered signal
import IPython.display as ipd
print("filtered signal 1, high-pass filter f>500Hz")
ipd.display(ipd.Audio(s_filtered1/10,rate=fs))
print("filtered signal 2, low-pass filter f<500Hz")
ipd.display(ipd.Audio(s_filtered2/10,rate=fs))
print("filtered signal 3, band-stop filter 300Hz>f and  f>1100Hz")
ipd.display(ipd.Audio(s_filtered3/10,rate=fs))
print("filters signal 4, band-pass filter 300Hz< f <3400Hz")
ipd.display(ipd.Audio(s_filtered4/10,rate=fs))
```

Question Q2:

- Explain the behaviour of the designed band-stop filter, i.e. describe (briefly) what you can see in the generated plots. If you didn't generate plots you can explain what you would expect to see.

Answer to Question Q2:

The band-stop filter plot shows that the signal that is being processed starts from level 1 up to 300Hz, then goes to 0 unti it reaches 1100Hz and finally it goes up to 1 until it reaches infinity. That means that the frequencies below 300Hz and above 1100Hz are multiplied by one and the frequencies in the middle, they are multiplied by 0. If we look at the spectogram, we can see that there are shades of light green and blue. At the bottom of the spectogram we can spot a blue horizontal line that shows that there is no signal in that frequency. This line is equal to the frequency between 300Hz and 1100Hz.

Question Q3:

- Which sounds are most affected when the low-pass cut-off frequency is set to around 500 Hz - vowels or consonants - and why?

Answer to Question Q3:
As a general observation, vowels tend to be lower in frequency comparing to consonants. Looking at the low-pass filter we notice that the vowels keep almost the same quality of sound while the consonants are being filtered and their quality deteriorates. As a result the consonants end up sounding less clear comparing to the vowels. On the other hand, in the case of the high-pass filter, we end up with the exact opposite effect; The consonants sound clearer while it's getting more difficult to distinguish the sound of the vowels.

4. Audio Effects

4.1 Low Frequency Oscillator

Many 'Voice effects (FX)' are achieved by modifying some characteristic of the speech using a low frequency oscillator or *LFO*. LFOs typically have two controls: speed (which is specified by the frequency in Hertz) and depth (which specifies the magnitude of the effect). The following tasks will require several LFOs, so it makes sense to implement one in the following.

Task T3:

- Implement a Low Frequency Oscillator as a function `lfo()` as described below. Visualise that your function works be generating a sine and a square wave of frequency 5 Hz and length 2 seconds with different depths.

Note: There will be an extra point in the marking if you **do not** use the `scipy` library to solve this task.

In [ ]:

```
def generateSquare(f0=1, length = 6, fs=44100, order=10):
    t_square=np.arange(0,length,1/fs)        # time vector
    sum = np.zeros(len(t_square))            # pre-allocate variable with zeros
    for ii in range(1, order+1, 2):
        sum += np.sin(2*np.pi*ii*f0*t_square)/ii
        #print(str(ii)+': adding sin(2 $\pi$ '+str(ii*f0)+' Hz t)')
    return 4/np.pi*sum, t_square

f0=1                                         # desired frequency in Hz
rec,t_square = generateSquare(f0,order=100)
plt.plot(t_square,rec,label='square Fourier');
```

In [ ]:

```
from scipy import signal as sig         # filter designs
t_t3 = np.linspace(0, 2, num = 88200)

def lfo1(speed_hz, depth, num_samples, fs=44100, square_curve=False):
    '''
    Low-frequency oscillator

    Parameters
    ----------
    speed_hz : float
       frequency of generated signal in Hertz
    depth : float
        magnitude of the effect
    num_samples : int
        length of the signal in samples
    fs : float, optional
        sampling frequency in Hz, default 44100
    square_curve : boolean, optional (default: False)
        generate square wave if true, generate sine wave if false

    Example use:
    -------
        sig_square = lfo(speed_hz=5, depth=0.7, num_samples=88200, fs=44100, square_curve=True)
    '''

    # Your code here
    if square_curve==True:
        r,s1= generateSquare(speed_hz,2,fs,10)
        r=r*depth
    else:
        s1 = depth * np.sin(2*np.pi*speed_hz*t_t3)
        r=0
    return (r,s1,sig)
r, t_square,whatever = lfo1(speed_hz=5, depth=2, num_samples=88200, fs=44100, square_curve=True)
r1,s1,whatever= lfo1(speed_hz=5, depth=2, num_samples=88200, fs=44100, square_curve=False)
fig=plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.plot(t_t3,r )
plt.title('Visualisation of the square filter ')
plt.xlabel('$t$ in seconds');
plt.subplot(1,2,2)
plt.plot(t_t3,s1)
plt.title('Visualisation of the sine filter ')
plt.xlabel('$t$ in seconds');
```

In [ ]:

```
from scipy import signal as sig         # filter designs

t = np.linspace(0, 2, num = 88200)

def lfo(speed_hz, depth, num_samples, fs=44100, square_curve=False):
    '''
    Low-frequency oscillator

    Parameters
    ----------
    speed_hz : float
       frequency of generated signal in Hertz
    depth : float
        magnitude of the effect
    num_samples : int
        length of the signal in samples
    fs : float, optional
        sampling frequency in Hz, default 44100
    square_curve : boolean, optional (default: False
        generate square wave if true, generate sine wave if false

    Example use:
    -------
        sig_square = lfo(speed_hz=5, depth=0.7, num_samples=88200, fs=44100, square_curve=True)
    '''
    if square_curve==True:
        signal1 = depth * sig.square(2 * np.pi * speed_hz * t)
    else:
        signal1 = depth * np.sin(2*np.pi*speed_hz*t)
    return (signal1)

sig_square = lfo(speed_hz=5, depth=0.7, num_samples=88200, fs=44100, square_curve=True)
sig_sin = lfo(speed_hz=5, depth=0.7, num_samples=88200, fs=44100, square_curve=False)

fig=plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.plot(t,sig_square)
plt.title('Visualisation of the square filter ')
plt.xlabel('$t$ in seconds');

plt.subplot(1,2,2)
plt.plot(t,sig_sin)
plt.title('Visualisation of the sine filter ')
plt.xlabel('$t$ in seconds');
```

Although your function outputs audio, you are unlikely to be able to hear it as the frequency is so low. However, you can check that it is functioning correctly by combining is with other audio signals as we will do in the following.

4.2 Amplitude Modulation - Tremolo

Tremolo is one of the most basic voice manipulations that makes use of an LFO. In this effect, the amplitude of a speech signal is modulated (https://en.wikipedia.org/wiki/Amplitude_modulation), i.e. the speech waveform is multiplied by a variable gain that ranges between 0 and 1.

Your LFO outputs an audio signal between `-depth` and `+depth`. So, in order to modulate the amplitude of the speech correctly, the output of the LFO has to be scaled appropriately to range between 0 and 1.

Task T4:

- Implement a function `tremolo()` using your function `lfo()` and modulate the amplitude of the speech signal.
- Experiment with different settings for `speed` and `depth`. In particular, note that a square wave with speed between 3 and 4 Hz (and depth = 1) has a very destructive effect on the intelligibility of the output. This is because $3-4$ Hz corresponds to the typical syllabic rate of speech.
- Proof that the effect works by a proper visualisation of the filtered speech signal and describe what can be observed and perceived.

In [ ]:

```
def tremolo(signal, fs, speed_hz, depth, square_curve=False):
    t=np.linspace(0,len(signal)/fs,len(signal))
    signal2 = lfo(speed_hz=speed_hz, depth=depth, num_samples=len(signal), fs=fs, square_curve=square_curve)
    scaled_signal2 = 0.5 + (1/(2*depth)) * signal2 #scale signal so it ranges ron 0 to 1
    modulated_signal = np.multiply(scaled_signal2,signal)
    return (modulated_signal)
t=np.linspace(0,len(s)/fs,len(s))


#experiment with different settings for speed
#the depth of the LFO is irrelevant as the signal get scaled to range from 0 to 1
print("tremolo effect, square LFO, speed = 300Hz")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=200, depth=1, square_curve=True)/10,rate=fs))
print("tremolo effect, sine LFO, speed = 300Hz")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=200, depth=1, square_curve=False)/10,rate=fs))
print("tremolo effect, square LFO, speed = 3.5Hz")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=3.5, depth=1, square_curve=True)/10,rate=fs))
print("tremolo effect, sine LFO, speed = 3.5Hz")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=3.5, depth=1, square_curve=False)/10,rate=fs))

# Proof that the effect works by a proper visualisation of the filtered speech signal

start_sample = int(7*fs);                   # start at 7 sec
no_of_samples = 10000;
end_sample   = start_sample + no_of_samples;
sample_vec = np.linspace(start_sample, end_sample, no_of_samples)

fig=plt.figure(figsize=(8,7.5))

plt.subplot(3,1,1)
x=s[start_sample:end_sample];
t_sample=t[start_sample:end_sample];

plt.plot(t_sample,x,)
plt.xlabel('$t$ in seconds')
plt.title('original signal - 7.00sec to 7.25sec')


s1=tremolo(signal=s, fs=fs, speed_hz=20, depth=1, square_curve=False)
x1=s1[start_sample:end_sample];

plt.subplot(3,1,2)
plt.plot(t_sample,x1,)
plt.xlabel('$t$ in seconds')
plt.title('tremolo effect, sine LFO, speed = 20Hz - 7.00sec to 7.25sec')
plt.tight_layout()

s2=tremolo(signal=s, fs=fs, speed_hz=20, depth=1, square_curve=True)
x2=s2[start_sample:end_sample];

plt.subplot(3,1,3)
plt.plot(t_sample,x2,)
plt.xlabel('$t$ in seconds')
plt.title('tremolo effect, square LFO, speed = 20Hz - 7.00sec to 7.25sec')

plt.tight_layout()

print('To demonstrate that the the tremolo effect function works the graphs show the original signal as well as two modulated s
print( )
print('The first modulated signal results from a sine wave LFO of 20Hz speed. The amplitude of the modulated signal follows a s
print( )
print('The second modulated signal results from a square wave LFO of 20Hz speed. The  modulated signal consists of plateaus of
```

4.3 Ring Modulation

Another basic effect is to multiply the speech signal by the output of an LFO. This is known as 'ring modulation'.

Note: In the BBC TV series Dr. Who (https://en.wikipedia.org/wiki/Doctor_Who), the voices of the alien Daleks (https://en.wikipedia.org/wiki/Dalek) are generated by a ring modulator with an LFO set to around 30 Hz. The voice actors also spoke using a stilted monotonic intonation in order to enhance the effect. You can try this yourself by recording your own voice and applying the effect.

Task T5 (Ring Modulation):

- Implement a function `ring_modulation()` using your function `lfo()` and modulate the amplitude of the speech signal by multiplying with the LFO signal.
- Experiment with different settings for `speed` and `depth` . Note how the timbre of the resulting sound is subtly different from *tremolo*.
- Proof that the effect works by a proper visualisation of the filtered speech signal and describe what can be observed and perceived.

In [ ]:

```
def ring_modulation(signal, fs, speed_hz, depth, square_curve=False):
    t=np.linspace(0,len(signal)/fs,len(signal))
    signal2 = lfo(speed_hz=speed_hz, depth=depth, num_samples=len(signal), fs=fs, square_curve=square_curve)
    modulated_signal = np.multiply(signal2,signal)
    return (modulated_signal)
t=np.linspace(0,len(s)/fs,len(s))


#experiment with different settings for speed and depth

print("tremolo effect, square LFO, speed = 300Hz, depth=0.7")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=200, depth=0.7, square_curve=True)/10,rate=fs))
print("tremolo effect, sine LFO, speed = 300Hz, depth=0.7")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=200, depth=0.7, square_curve=False)/10,rate=fs))
print("tremolo effect, square LFO, speed = 3.5Hz , depth=0.2")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=3.5, depth=0.2, square_curve=True)/10,rate=fs))
print("tremolo effect, sine LFO, speed = 3.5Hz, depth=0.2")
ipd.display(ipd.Audio(tremolo(signal=s, fs=fs, speed_hz=3.5, depth=0.2, square_curve=False)/10,rate=fs))
```

In [ ]:

```
# Your code here to show an example of the Ring Modulation effect
%matplotlib inline
import matplotlib.pyplot as plt  # plotting
import numpy as np               # math

#read soundfile that is already downloaded
import soundfile as sf
s_file_name = 'speech.wav'
s,fs = sf.read(s_file_name)

from scipy import signal as sig         # filter designs
import scipy.signal as sig

def nextpow2(n):
    return int(np.ceil(np.log2(np.abs(n))))

def frequency_shift(signal, fs, shift_amount):
    N_orig = len(signal)
    N_padded = 2 ** nextpow2(N_orig)
    t = np.arange(0, N_padded)
    return (sig.hilbert(np.hstack((signal, np.zeros(N_padded - N_orig, signal.dtype))))* np.exp(2j * np.pi * shift_amount * t /


signal_shifted = frequency_shift(s, fs, 10000)


# calculate the spectum (frequency domain representation)
FFT_length = 2**15 # take a power of two which is larger than the signal length
f = np.linspace(0, fs/2, num=int(FFT_length/2+1))
spectrum = np.abs(np.fft.rfft(s,n=FFT_length))
# plot the spectrum
plt.subplot(2,1,1)
plt.plot(f,spectrum)
plt.title('Visualisation of the signal in spectral domain')
plt.xlabel('frequency $f$ in Hz')
plt.ylabel('$x(f)$')

spectrum1 = np.abs(np.fft.rfft(signal_shifted,n=FFT_length))
plt.subplot(2,1,2)
plt.plot(f,spectrum1)
plt.title('Visualisation of the signal in spectral domain')
plt.xlabel('frequency $f$ in Hz')
plt.ylabel('$x(f)$')

plt.tight_layout()
```

4.4 Frequency Shifting

Many Vocal FX are the result of altering the frequencies present, e.g. changing the pitch of a voice. There are many algorithms for frequency shifting. You have already implemented an approximate solution with your ring modulator.

For simplicity, the following function will be givem implementing frequency shifting.

In [ ]:

```
# the following code in this cell is taken and slightly adapted from:
# https://gist.github.com/lebedov/4428122

import scipy.signal as sig

def nextpow2(n):
    '''Return the first integer N such that 2**N >= abs(n)'''
    return int(np.ceil(np.log2(np.abs(n))))

def frequency_shift(signal, fs, shift_amount):
    '''
    Shift the specified signal by the specified frequency.

    Parameters
    ----------
    signal : float
       input signal to which the effect should be applied
    fs : int
        sampling frequency in Hz
    shift_amount : float
       amount of frequency shift (in Hz)

    Return
    ----------
    signal after application of the frequancy shifting effect

    Example use:
    -------
        signal_frequency_shifted = frequency_shift(audio_in, fs, 100)
    '''

    # Pad the signal with zeros to prevent the FFT invoked by the transform from
    # slowing down the computation:
    N_orig = len(signal)
    N_padded = 2 ** nextpow2(N_orig)
    t = np.arange(0, N_padded)
    return (
        sig.hilbert(
            np.hstack((signal, np.zeros(N_padded - N_orig, signal.dtype)))
        )
        * np.exp(2j * np.pi * shift_amount * t / fs)
    )[:N_orig].real
```

Task T6 (Frequency Shifting):

- Visualise the effect of the frequency shift effect using an appropriate spectral representation.

In [ ]:

```
%matplotlib inline

def nextpow2(n):
    return int(np.ceil(np.log2(np.abs(n))))

def frequency_shift(signal, fs, shift_amount):
    N_orig = len(signal)
    N_padded = 2 ** nextpow2(N_orig)
    t = np.arange(0, N_padded)
    return (sig.hilbert(np.hstack((signal, np.zeros(N_padded - N_orig, signal.dtype))))* np.exp(2j * np.pi * shift_amount * t /

signal_shifted = frequency_shift(s, fs,100)
# calculate the spectum (frequency domain representation)
FFT_length = 2**15 # take a power of two which is larger than the signal length
f = np.linspace(0, fs/2, num=int(FFT_length/2+1))
spectrum = np.abs(np.fft.rfft(s,n=FFT_length))
spectrum_shifted = np.abs(np.fft.rfft(signal_shifted,n=FFT_length))

# plot the spectra
f_sample=f[0:800]
spectrum_sample=spectrum[0:800]
spectrum_shifted_sample=spectrum_shifted[0:800]
plt.plot(f_sample,spectrum_sample,label='original signal')
plt.plot(f_sample,spectrum_shifted_sample,'r',label='100Hz shift')
plt.title('Visualisation of the signal in spectal domain')
plt.xlabel('frequency $f$ in Hz')
plt.ylabel('$x(f)$')
plt.legend()
```

In [ ]:

```
ipd.Audio(s/10,rate=fs)
```

In [ ]:

```
ipd.Audio(signal_shifted/10,rate=fs)
```

Question Q4:

- COM3502-4502-6502: Why can the voice be shifted up in frequency much further than it can be shifted down in frequency before it becomes severely distorted? Hint: Calculate a spectrum plot if the answer is not immediately clear to you.
- COM4502-6502 ONLY: Your frequency shifter changes all the frequencies present in an input signal. How might it be possible to change the pitch of a voice without altering the formant frequencies?

Answer to Question Q4:

First part of the question: The lower frequencies occupy the biggest part of the voice energy. If the frequencies that occupy the voice are being shifted from high to low, then the lowest frequency is being removed. That means that the lower the frequency when shifting the signal, the more the voice will be distorted. However, going up in frequency then the voice is not that severely affected as the frequencies can go higher without removing the lower frequencies, which are the ones that give more power, energy and value to the voice. Second part of the question: This could be possible by modifying the formant frequancies of the signal in order to apply another energy distribution on the same frequencies by modulating (apply tremolo effect for example) or filtering the signal.

4.5 Harmony Effect

A classic 'robotic' voice can be achieved by simply adding frequency-shifted speech back to the unprocessed original. This effect is known as 'harmony'. However, rather than simply adding the signals in equal amounts, we will implement a more general purpose approach.

Task T7:

- Implement a function `mixer()` that adds the original speech with the manipulated speech in different proportions.
- Implement a function `harmony()` mixes the input signal with a frequency shifted version of itself (using the funtions `mixer()` and `frequency_shift()`). With your mixer at the 50-50 setting, experiment with different frequency shifts in order to produce the best robotic sounding output. Report "your optimal" setting.

In [ ]:

```
# def mixer(signal1, signal1, percentage_1=0.5):

# Your code here to implement mixing of two signals, used later for the harmony effect

def mixer(signal1, signal2, percentage_1=0.5):
    return(percentage_1*signal1+(1-percentage_1)*signal2)
```

In [ ]:

```
# def harmony(...

# Your code here to implement the harmony effect

def harmony(signal1, shift_amount, percentage_1=0.5):
    signal2=frequency_shift(signal1, fs,shift_amount)
    return(mixer(signal1=signal1,signal2=signal2, percentage_1=percentage_1))
```

In [ ]:

```
ipd.Audio(harmony(s,350,0.5)/10,rate=fs)
```

Answer to question in Task T7:

My optimal setting as represented above by the audio file is 350,0.5/10
because the sound that is being produced simulates that of a robot without producing a cacophony that would potentially lead to distorted, unrecognisable and unidentifiable sounds.

4.6 Frequency Modulation: Vibrato

Now that you have the ability to shift the frequencies in a speech signal, it is very easy to implement another common voice manipulation technique - *vibrato*. All that is required is for the frequency shifter to be controlled by the output of an LFO.

Task T8:

- Implement a function `vibrato()` by connecting an LFO to your frequency shifter, and experiment with different values for speed and depth . Note that the LFO output will need to be scaled to provide an appropriate frequency shift range and then added to the output of the frequency shift.

In [ ]:

```
from scipy import signal as sig         # filter designs

def lfo(speed_hz, depth, num_samples, fs=44100, square_curve=False):
    t = np.linspace(0, num_samples/fs, num = num_samples)
    if square_curve==True:
        signal1 = depth * sig.square(2 * np.pi * speed_hz * t)
    else:
        signal1 = depth * np.sin(2*np.pi*speed_hz*t)
    return (signal1,t)
```

In [ ]:

```
!curl https://staffwww.dcs.shef.ac.uk/people/S.Goetze/sound/speech.wav -o speech.wav

import soundfile as sf
s_file_name = 'speech.wav'
s,fs = sf.read(s_file_name)

def vibrato(signal,shift_amount,LFO_speed_hz,LFO_square_curve=False,fs=44100):
    lfo1,t = lfo(speed_hz=LFO_speed_hz, depth=1, num_samples=1048576, fs=fs, square_curve=LFO_square_curve)
    scaled_lfo1 = 0.5 +  0.5*lfo1 #scale signal so it ranges from 0 to shift range

    shift_amount1=shift_amount*scaled_lfo1
    return(frequency_shift(signal, fs, shift_amount1))

x=vibrato(s,20,LFO_speed_hz=5)
ipd.Audio(x/10,rate=fs)
```

4.7 Time Delay Effect - Echo and Comb Filter

Many interesting voice FX can be achieved by delaying the signal and recombining it with itself.

Task T9:

- Implement a function `echo()` which mixes a signal $s(t)$ with itself in a delayed version, i.e. $s(t - t_0)$. Experiment with various values for the delay t_0, and note the different effects you can achieve with delays
  - below 20 msecs
  - between 20 and 100 msecs, and
  - above 100 msecs.

In [ ]:

```
# def echo(...

# Your code here to show an example of the echo effect
#below 20ms you cant hear the echo since both signals are too close.
#at 100ms you hear both sounds clear--echo
t=np.linspace(0,len(s)/fs,len(s))

def echo(signal, delay):
    startsample=int(delay/(1/fs))
    print(startsample)
    numofsamples=int(len(s));
    print(numofsamples)
    endsample=numofsamples;
    print(endsample)
    samplevec=np.linspace(startsample,endsample,numofsamples-startsample);
    s1=s[startsample:numofsamples];
    s2=s[0:numofsamples-startsample]
    xecho=s2+s1;

    return(samplevec,xecho);

sv,xe=echo(s,0.01)   # below 20 msecs
plt.plot(sv,xe)
plt.xlabel('$t$');
plt.ylabel('$s(t)$');
plt.title("Time delay effect using echo filter(10 msecs)")

ipd.Audio(xe, rate=fs)
```

In [ ]:

```
sv,xe=echo(s,0.05)   # between 20 and 100 msecs
plt.plot(sv,xe)
plt.xlabel('$t$');
plt.ylabel('$s(t)$');
plt.title("Time delay effect using echo filter(50 msecs)")

ipd.Audio(xe, rate=fs)
```

In [ ]:

```
sv,xe=echo(s,0.5)  # above 100 msecs
plt.plot(sv,xe)
plt.xlabel('$t$');
plt.ylabel('$s(t)$');
plt.title("Time delay effect using echo filter(500 msecs)")

ipd.Audio(xe, rate=fs)
```

4.8 Comb Filtering

You should observe that, with delays below 20 msec in your function `echo()`, the signals combine to create a subtle 'phasing' effect. This is known as 'comb filtering' as the signal is effectively interfering with itself, and frequency components corresponding to multiples of the delay time are enhanced or cancelled out (due to 'superposition'). Delays between 20 and 100 msecs give the effect of the voice being in a reverberant room. Delays above 100 msecs sound like distant echoes.

Task T10:

- Visualise the impulse response of your function `echo()` as well as the transfer function. Can you give an explanation from havin a look at the transfer function, why this effect would be called *comb filter*?

In [ ]:

```
# impulse response at 100Hz

a= signal.unit_impulse(44100,441)
print(a.shape)
fs3=441000
dt3=1/fs3
stoptime=0.5
ta=np.arange(0,stoptime,dt3)
print(ta.shape)
#plt.plot(ta,a)
plt.plot(a)
plt.title("Impulse Response")
plt.xlabel('k')
plt.ylabel('amplitude')
```

In [ ]:

```
from scipy import signal

def echo2(signal1, delay):
    fs5 = 44100                          # samples per second
    startsample=int(delay/(1/fs5))
    print(startsample)
    numofsamples=int(len(signal1));
    print(numofsamples)
    endsample=numofsamples;
    print(endsample)
    samplevec=np.linspace(0,endsample,numofsamples);
    s11=np.roll(signal1, startsample)
    s21=signal1
    xecho=s21+s11;

    return(samplevec,xecho);

sv2,xe2=echo2(a,0.01)#value smaller than 20ms
plt.plot(sv2,xe2);
plt.title("Impulse Response with Echo(10msec delay)")
plt.xlabel('k')
plt.ylabel('amplitude')
```

In [ ]:

```
fs5=44100
FFT_length = 2**15
freq5= np.linspace(0, fs5/2, num=int((FFT_length/2)+1))#we only plot half of the fft lenght (the positive half)--the other hal
spectrum2= np.abs(np.fft.rfft(xe2,FFT_length))
print(spectrum2.shape)
print(freq5.shape)

plt.plot(freq5[0:2000],spectrum2[0:2000])
plt.title("Spectrum for Echo function(comb effect)")
plt.xlabel('k')
plt.ylabel('amplitude')
```

Answer to question in Task T10:
After using a small frequency and zoom in to that frequency, it's noticable that the frequencies have the shape of a comb. That is evident from the plot above.

4.9 Flanger

It is possible to use an LFO to vary the delay. The resulting effect is known as a *flanger*.

Task T11:

- Add an LFO to your 'delay' to create a 'flanger', and experiment with different settings. Note that you will need to scale the output of the LFO, and you will get different effects depending on whether the delayed signal is mixed with the original or not.

In [ ]:

5 Frequency Analysis

Question Q5:

- COM3502-4502-6502:
  - What does FFT stand for and what does an FFT do?
- COM4502-6502 ONLY: What is a DFT and how is it different from an FFT?

Answer to Question Q5:
First question: FFT is an abbreviation of Fast Fourier transform. More specifically, it is a mathematical algorithm used in signal processing that makes conversions made by DFT (abbreviation for Discrete Fourier Transform) faster and more efficient by reducing the number of computations needed. Second question: DFT is also a mathematical algorithm that contributes in processing digital signals by calculating the spectrum of a finite-duration signal. DFT works by transforming N discrete-time domain samples of signals to the frequency domain components. In some applications, the shape of the time domain is not applicable for signals. In occasions like these signal frequency content becomes very useful. FFT is an implementation of DFT whilst DFT illustrates a relationship between the time domain and the frequency domain representation. DFT is a mathematical algorithm that converts time-domain signals to frequency domain components, while in contrast, FFT algorithm involves several computation techniques including DFT.

Creating the Spectrogram Step-by-Step (for COM4502-6502 only)

The magnitude $|X[n,\ell]|$ of the STFT for all n and ℓ is known as the spectrogram (https://en.wikipedia.org/wiki/Spectrogram) of a signal. It is frequently used to analyze signals in the time-frequency domain, for instance by a spectrum analyzer (https://en.wikipedia.org/wiki/Spectrum_analyzer). It can be interpreted as a *image* of the signal with (block) time direction on the x axis and (discrete) frequency n on the y axis.

From Lab Sheet 3 we already know how to brack a long signal into block, a.k.a. frames.

Task 12: Manual Spectrogram Calculation (for COM4502-6502 only)

- Implement a function `calc_SpectralPoint(xk,n)` which calculates a spectral point for one discrete frequency n from a input frame $x[k]$, i.e. a function which implements the well-known DFT equation
$$\mathrm{DFT}\{x[k]\} = X[n] = \frac{1}{L_{\mathrm{DFT}}} \sum_{k=0}^{L_{\mathrm{DFT}}-1} x[k] e^{j2\pi kn/L_{\mathrm{DFT}}}$$
for one fixed n.
- Implement a very similar function `calc_SpectralPointWindowed(xk,n)` which calculates a spectral point for one discrete frequency n from a input frame $x[k]$, but in addition applies a window function $w[k]$ to the frame $x[k]$, i.e. the function should calculate
$$\mathrm{DFT}\{w[k] \cdot x[k]\} = X^{\mathrm{w}}[n] = \frac{1}{L_{\mathrm{DFT}}} \sum_{k=0}^{L_{\mathrm{DFT}}-1} w[k] x[k] e^{j2\pi kn/L_{\mathrm{DFT}}}$$
for one fixed n. The window should have the same length L_{DFT} as your frame and should be one of the windows we discussed during the lecture.
- The functions above only calculates one spectral value at a time. To obtain a full spectrum , implement a function `calc_Manitude_Spectrum()` which transforms every windowed frame to the frequency domain and calculates all positive frequencies, i.e. for $0 \leq n \leq L_{\mathrm{DFT}}/2 + 1$.
- Create a function `create_spectrogram()` , which splits the complete input sequence (e.g. a loaded WAVE file) into blocks of length L_{DFT}. These may be overlapping. For each block the spectrum should be calculated using the previously implemented function `calc_Manitude_Spectrum()` and all spectra should be collected to form a spectrogram (e.g. as columns of a matrix).
- Concatenate the resulting spectra to a spectrogram and display the resulting spectrogram. You can use `matplotlib` 's `imshow()` (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html) function for manually plotting the spectrogram image. Note that a spectrogram is usally shown in dB scaling.
- Visulalise the input signal $x[k]$ as spectrogram for (i) a speech signal and (ii) for a chirp/sweep signal.

Implement the DFT equation

$$\mathrm{DFT}\{x[k]\} = X[n] = \frac{1}{L_{\mathrm{DFT}}} \sum_{k=0}^{L_{\mathrm{DFT}}-1} x[k] e^{j2\pi kn/L_{\mathrm{DFT}}}$$

for one fixed n:

In [ ]:

```
#calculate spectral point for discrete frequency 'n'
def calc_SpectralPoint(xk,n):
    sum=0
    for k in range(len(xk)):
        sum =sum+ xk[k]*np.exp(2j*np.pi*k*n/len(xk))
    Xn=sum/len(xk)
    return(Xn)
```

Implement DFT of windowed frame

$$\mathrm{DFT}\{w[k]\cdot x[k]\} = X^{\mathrm{w}}[n] = \frac{1}{L_{\mathrm{DFT}}}\sum_{k=0}^{L_{\mathrm{DFT}}-1} w[k]x[k]e^{j2\pi kn/L_{\mathrm{DFT}}}$$

for one fixed n.

In [ ]:

```
def calc_SpectralPointWindowed(xk,n,window=False):
    sum=0
    win = np.ones(len(xk))
    a=np.empty(len(xk))
    if window==True:
        for i in range(len(xk)):
            win[i]=0.5*(1-np.cos(2*np.pi*i/len(xk)))
    for d in range(len(xk)):
        sum = sum + win[d]*xk[d]*np.exp((2j*np.pi*d*n)/len(xk))
    Xwn=sum/len(xk)
    return(Xwn)
calc_SpectralPointWindowed(s,200,window=True)
```

The following function `calc_Manitude_Spectrum()` is supposed to transform every (windowed or not windowed) frame to the frequency domain and to calculate all positive frequencies, i.e. $X[n]$ or $X^{\mathrm{w}}[n]$ for $0 \leq n \leq L_{\mathrm{DFT}}/2+1$.

The following function `create_spectrogram()` should calculate all spectra needed for your spectrogram.

In [ ]:

```
def calc_Manitude_Spectrum(xk):
    FFT_length = 2**15 # take a power of two which is larger than the signal length
    f = np.linspace(0, fs/2, num=int(FFT_length/2+1))
    spectrum = np.abs(np.fft.rfft(xk,n=FFT_length))
    # plot the spectrum
    plt.plot(f,spectrum)
    plt.title('Visualisation of the signal in spectal domain')
    plt.xlabel('frequency $f$ in Hz')
    plt.ylabel('$x(f)$')
    return(spectrum)

calc_Manitude_Spectrum(s)
```

In [ ]:

```
def create_spectrogram(x, L_DFT=512, noverlap):
    '''
          x: original time series
      L_DFT: The number of data points used in each block for the DFT. The default value is 512.
    noverlap: The number of points of overlap between blocks. The default value is 256.
    '''
    # Your code here
    # ...
```

The following function can be used to actually display the created spectrogram.

In [ ]:

```
def plot_spectrogram( #...
```

The following code actually calculates and plots your spectrogram (for the two signals mentioned above). Feel free to adapt parameters `L_DFT` and `noverlap`

In [ ]:

```
# load or create signal

# create and plot spectrogram (generated using your functions above)
L_DFT    = 256 # DFT length
noverlap = 84  # number of overlapping samples
starts, spec = create_spectrogram( #...
plot_spectrogram(#...
```

6 Equaliser (for COM4502-6502 only)

We want to design an equaliser like shown in the picture below as a hardware system.

Picture taken from Wikipedia (https://simple.wikipedia.org/wiki/Equalization_(audio)), license: CC BY 2.0 (https://creativecommons.org/licenses/by/2.0/)

The following function realises one of the sliders in software.

In [ ]:

```
def peaking_filter(gain,center_freq,q,fs):
    """
    Derive coefficients for a peaking filter with a given amplitude and
     bandwidth.  All coefficients are calculated as described in Zolzer's
     DAFX book (p. 50 - 55).  This algorithm assumes a constant q-term
     is used through the equation.

    Usage:     `b,a` = peaking_filter(gain,center_freq, q,fs)
                `gain` is the logrithmic gain (in dB)
                `center_freq` is the center frequency
                `q` is q-term equating to (Fb / Fc)
                `fs` is the sampling rate

    Author:    Jeff Tackett 08/22/05
    Port to Python by George Close 10/07/21
    """

    gain = np.float32(gain)
    k = np.tan((np.pi*center_freq)/fs)
    V0 = 10**((gain)/20)
    # invert gain if a cut
    if V0 < 1:
        V0 = 1/V0

    # Boost
    if gain > 0:
        b0 = (1 + ((V0/q)*k)+ k**2) / (1+((1/q)*k)+k**2)
        b1 = (2 * (k**2 - 1)) / (1 + ((1/q)*k) + k**2);
        b2 = (1 - ((V0/q)*k) + k**2) / (1 + ((1/q)*k) + k**2);
        a1 = b1;
        a2 =  (1 - ((1/q)*k) + k**2) / (1 + ((1/q)*k) + k**2);
    # Cut
    elif gain <0:
        b0 = (1 + ((1/q)*k) + k**2) / (1 + ((V0/q)*k) + k**2);
        b1 =       (2 * (k**2 - 1)) / (1 + ((V0/q)*k) + k**2);
        b2 = (1 - ((1/q)*k) + k**2) / (1 + ((V0/q)*k) + k**2);
        a1 = b1;
        a2 = (1 - ((V0/q)*k) + k**2) / (1 + ((V0/q)*k) + k**2);
    #gain is 0
    else:
        b0 = V0;
        b1 = 0;
        b2 = 0;
        a1 = 0;
        a2 = 0;
    a = [  1, a1, a2];
    b = [ b0, b1, b2];
    return b,a
```

Task T13: (for COM4502-6502 only)

- Visualise the frequency response of one filter.
- Implement a cascade of filters to realise an equaliser.
- Visualise the frequency response of your equaliser filter and the input and (filtered) output signal.

In [ ]:

```
# we are using gain = 5, center frequency = 5000, q-factor = 2, sampling frequncy = 44100
b,a = peaking_filter(gain= 5,center_freq= 5000, q = 2,fs= 44100)
print(a,b)
f_eq,h_eq = signal.freqz(b,a)
omega_eq = np.linspace(0,1,len(f_eq))
# plot filter
plt.plot(omega_eq,np.abs(h_eq), lw=2, label='Equaliser')
plt.title("Frequency response of peaking filter")
plt.xlabel('f/fs')
plt.ylabel('Gain')
```

In [ ]:

```
# cascade filter
b1,a1 = peaking_filter(gain= 5,center_freq= 50, q = 2,fs= 44100)
b2,a2 = peaking_filter(gain= 5,center_freq= 500, q = 2,fs= 44100)
b3,a3 = peaking_filter(gain= 5,center_freq= 300, q = 2,fs= 44100)

f_eq1,h_eq1 = signal.freqz(b1,a1)
f_eq2,h_eq2 = signal.freqz(b2,a2)
f_eq3,h_eq3 = signal.freqz(b3,a3)
h_eq4 = h_eq1 + h_eq2 + h_eq3-3

plt.figure(figsize=(10,5))
plt.plot(omega_eq[0:100],np.abs(h_eq4[0:100]), lw=4, label='Equaliser')
plt.title("Cascade filters")
plt.xlabel('f/fs')
plt.ylabel('Gain')
```

In [ ]:

```
s_filtered11 = signal.filtfilt(b1, a1, s)
s_filtered22 = signal.filtfilt(b2, a2, s_filtered11)
s_filtered33 = signal.filtfilt(b3, a3, s_filtered22)

plt.plot(t,s_filtered33)
plt.title("Cascade filters")
plt.xlabel('f/fs')
plt.ylabel('Gain')

ipd.Audio(s_filtered33/10,rate=fs)
```

6. Prepare for submission

Task T14:

- Clear all cell outputs to reduce the file size (in Jupyter Notebooks clivk on "Cell->All Output->Clear")
- Create a `.zip` file named `YourName.zip` containing this Jupyter Notebook files as well as all other file necessary to run this notebook (**if such exist**, e.g. if you created (additional) WAVE files).
- Hand-in your `.zip` file via Blackboard.

Important: For marking, we expect your code to work 'out of the box'. This means that no additional software should have to be installed to make the Notebook run. If you only used libraries known from the Speech Processing Lab classes, you should be safe here.

In [ ]:

```
```