st121413-lab8

November 10, 2020

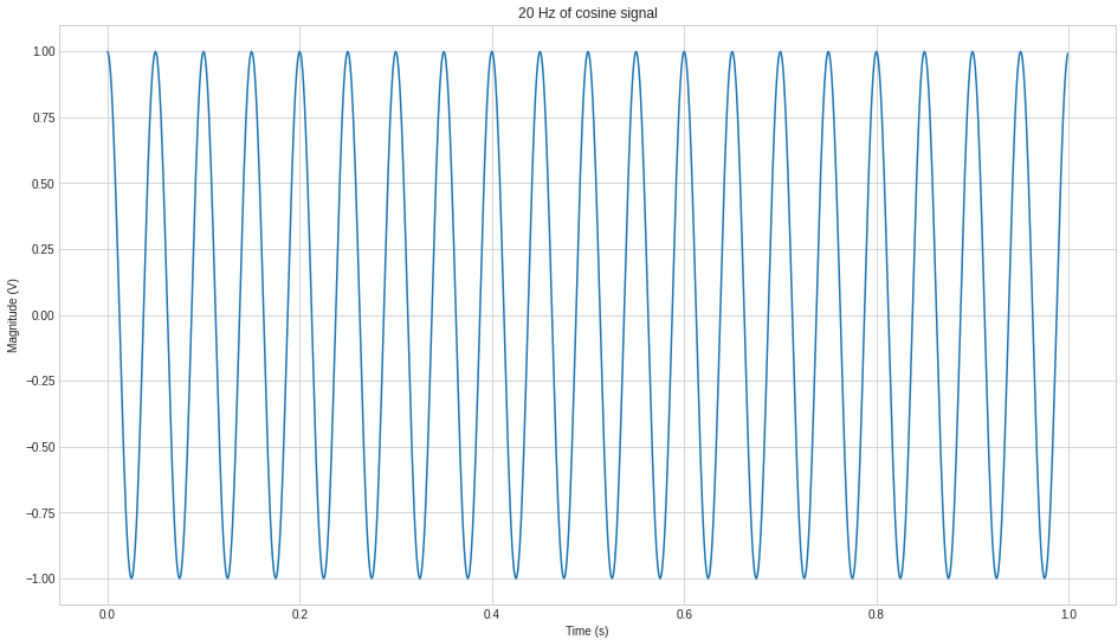
1 Lab 8 - st121413

1.1 Akraradet Sinsamersuk

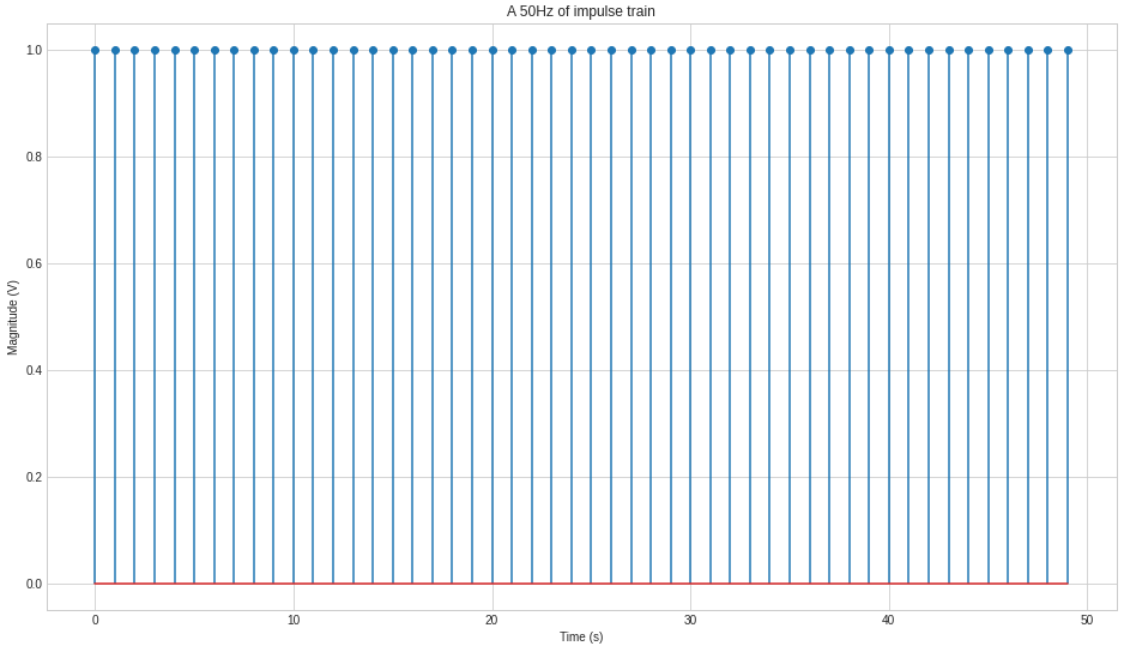
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as signal
from scipy.fftpack import fft, fftshift
plt.style.use('seaborn-whitegrid')


def calFFT(signal, window = 2048 , shift = False , inDB = False, half = True,␣
 ↪normf=True, fs=None):
    mag = np.abs(fft(signal, window) / (len(signal)/2.0))
    freq = np.linspace(0, 1, len(mag))

    if shift:
        mag = np.abs(fftshift(mag / abs(mag).max() ) )
        freq = np.linspace(-0.5, 0.5, len(mag))


    if inDB:
        mag = 20 * np.log10( mag )

    if normf == False:
        if fs == None:
            raise ValueError("Give me 'fs'")
        freq = np.linspace(0, fs, len(mag) )

    if half:
        mag = mag[:len(mag)//2]
        freq = freq[:len(freq)//2]



    return mag, freq
```

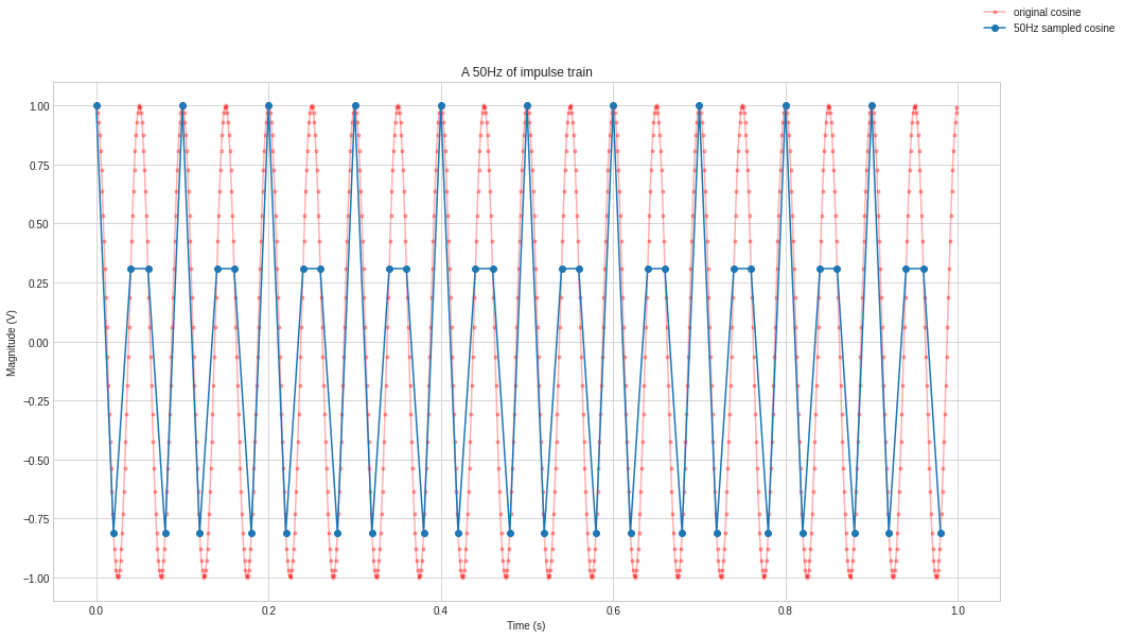
```
# DISCLAIMER: This function is copied from https://github.com/nwhitehead/
 ↪swmixer/blob/master/swmixer.py,
#             which was released under LGPL.
def resample_by_interpolation(signal, input_fs, output_fs):

    scale = output_fs / input_fs
    # calculate new length of sample
    n = round(len(signal) * scale)

    # use linear interpolation
    # endpoint keyword means than linspace doesn't go all the way to 1.0
    # If it did, there are some off-by-one errors
    # e.g. scale=2.0, [1,2,3] should go to [1,1.5,2,2.5,3,3]
    # but with endpoint=True, we get [1,1.4,1.8,2.2,2.6,3]
    # Both are OK, but since resampling will often involve
    # exact ratios (i.e. for 44100 to 22050 or vice versa)
    # using endpoint=False gets less noise in the resampled sound
    resampled_signal = np.interp(
        np.linspace(0.0, 1.0, n, endpoint=False),  # where to interpret
        np.linspace(0.0, 1.0, len(signal), endpoint=False),  # known positions
        signal,  # known data points
    )
    return resampled_signal
```

[2]:
```
# 1. Using the provided makecos function generate an analog signal with 20 Hz␣
 ↪frequency.
f = 20 #Hz
fs_original = 1000
t = np.linspace(0,1,fs_original, endpoint=False)
cos_20 = np.cos(2 * np.pi * f * t) # COS signal with 20Hz
fig, ax = plt.subplots(1, figsize=(16,9))
ax.plot(t,cos_20)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Magnitude (V)')
ax.set_title('20 Hz of cosine signal')
plt.show()
```



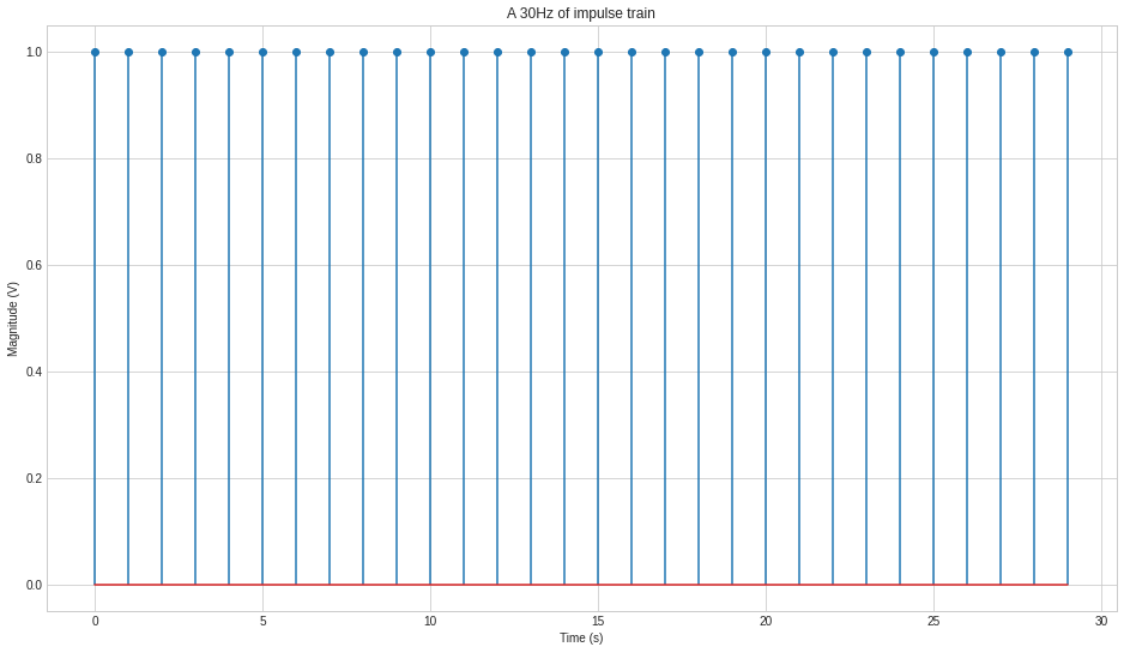
[3]:
```
# 2. Sample this signal using a 50 Hz(oversampling) sampling frequency.
# For this you have to generate an impulse train with frequency 50 Hz with the␣
 ↪given makeimp function then use sampleit1 function to sample the signal and␣
 ↪finally plot the signal with smpl plot function.

imp_50 = signal.unit_impulse(50, np.arange(0,50))
fig, ax = plt.subplots(1, figsize=(16,9))
ax.stem(imp_50)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Magnitude (V)')
ax.set_title('A 50Hz of impulse train')
plt.show()

cos_50 = resample_by_interpolation(cos_20, fs_original, 50)
t_50 = np.linspace(0, 1, 50, endpoint=False)
fig, ax = plt.subplots(1, figsize=(16,9))
ax.plot(t, cos_20, 'r.-', label = 'original cosine', alpha=0.3)
ax.plot(t_50,cos_50, 'o-', label = '50Hz sampled cosine')
fig.legend()
ax.set_xlabel('Time (s)')
ax.set_ylabel('Magnitude (V)')
ax.set_title('A 50Hz of impulse train')
plt.show()
```

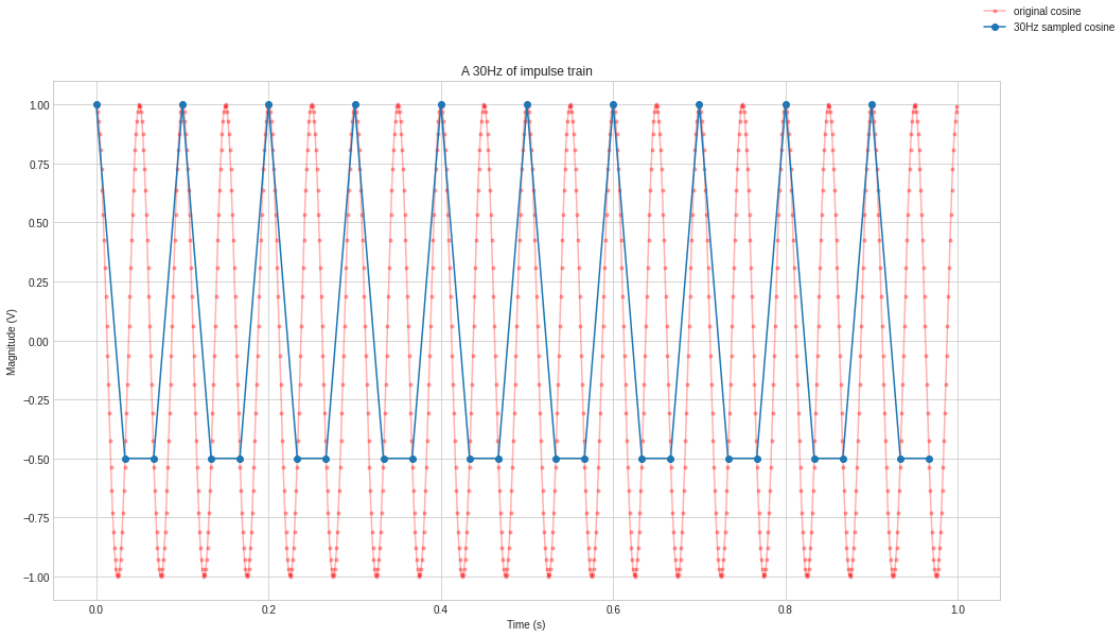




[15]:
```
# 3. Sample this signal using a 30 Hz(under sampling) sampling frequency. Use␣
↪the same
# steps as in the previous step.

imp_30 = signal.unit_impulse(30, np.arange(0,30))
```

```
fig, ax = plt.subplots(1, figsize=(16,9))
ax.stem(imp_30)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Magnitude (V)')
ax.set_title('A 30Hz of impulse train')
plt.show()

cos_30 = resample_by_interpolation(cos_20, fs_original, 30)
t_30 = np.linspace(0, 1, 30, endpoint=False)
fig, ax = plt.subplots(1, figsize=(16,9))
ax.plot(t, cos_20, 'r.-', label = 'original cosine', alpha=0.3)
ax.plot(t_30,cos_30, 'o-', label = '30Hz sampled cosine')
fig.legend()
ax.set_xlabel('Time (s)')
ax.set_ylabel('Magnitude (V)')
ax.set_title('A 30Hz of impulse train')
plt.show()
```

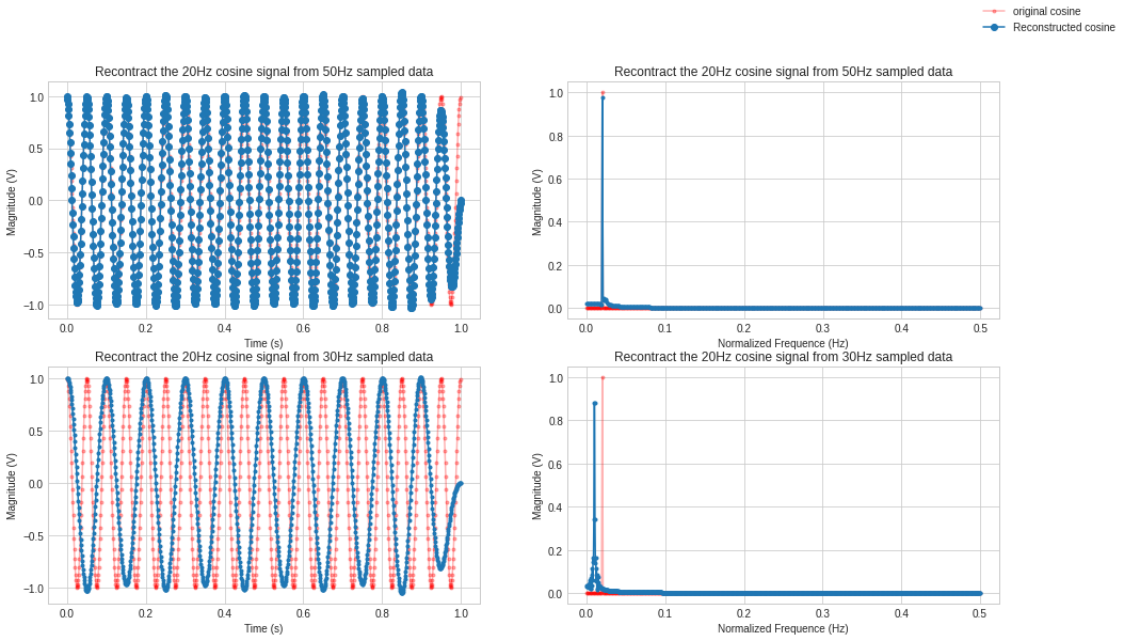




[5]:
```
# 4. Using the first sampled signal ms1, original signal can be reconstructed␣
 ↪while using ms2 aliasing will occur.

ms1 = cos_50
original = cos_20
ms2 = cos_30
```

[6]:
```
# 5. Reconstruct the signal using the function interpsinc function and plot the␣
 ↪waveforms in both time and frequency using recon plot function in order to␣
 ↪verify the sampling theory.


# Copy code from https://gist.github.com/endolith/1297227
def sinc_interp(x, s, u):
    """
    Interpolates x, sampled at "s" instants
    Output y is sampled at "u" instants ("u" for "upsampled")

    from Matlab:
    http://phaseportrait.blogspot.com/2008/06/sinc-interpolation-in-matlab.html␣
 ↪
    """
    import numpy as np
    if len(x) != len(s):
        raise ValueError('x and s must be the same length')
```

```
    # Find the period
    T = s[1] - s[0]

    sincM = np.tile(u, (len(s), 1)) - np.tile(s[:, np.newaxis], (1, len(u)))
    y = np.dot(x, np.sinc(sincM/T))
    return y

inter_t = np.linspace(0,1,1000)
inter_cos_50 = sinc_interp(cos_50, t_50, inter_t)

m1,f1 = calFFT(cos_20,None)
m2,f2 = calFFT(inter_cos_50,None)

fig, ax = plt.subplots(2,2, figsize=(16,9))
ax[0][0].plot(t, cos_20, 'r.-', label = 'original cosine', alpha=0.3)
ax[0][0].plot(inter_t,inter_cos_50, 'o-', label = 'Reconstructed cosine')
ax[0][0].set_xlabel('Time (s)')
ax[0][0].set_ylabel('Magnitude (V)')
ax[0][0].set_title('Recontract the 20Hz cosine signal from 50Hz sampled data')

ax[0][1].plot(f1, m1, 'r.-', alpha=0.3)
ax[0][1].plot(f2, m2, '.-')
ax[0][1].set_xlabel('Normalized Frequence (Hz)')
ax[0][1].set_ylabel('Magnitude (V)')
ax[0][1].set_title('Recontract the 20Hz cosine signal from 50Hz sampled data')

inter_cos_30 = sinc_interp(cos_30, t_30, inter_t)
m3,f3 = calFFT(inter_cos_30)

ax[1][0].plot(t, cos_20, 'r.-', alpha=0.3)
ax[1][0].plot(inter_t,inter_cos_30, '.-')
ax[1][0].set_xlabel('Time (s)')
ax[1][0].set_ylabel('Magnitude (V)')
ax[1][0].set_title('Recontract the 20Hz cosine signal from 30Hz sampled data')

ax[1][1].plot(f1, m1, 'r.-', alpha=0.3)
ax[1][1].plot(f3, m3, '.-')
ax[1][1].set_xlabel('Normalized Frequence (Hz)')
ax[1][1].set_ylabel('Magnitude (V)')
ax[1][1].set_title('Recontract the 20Hz cosine signal from 30Hz sampled data')



fig.legend()
plt.show()
```



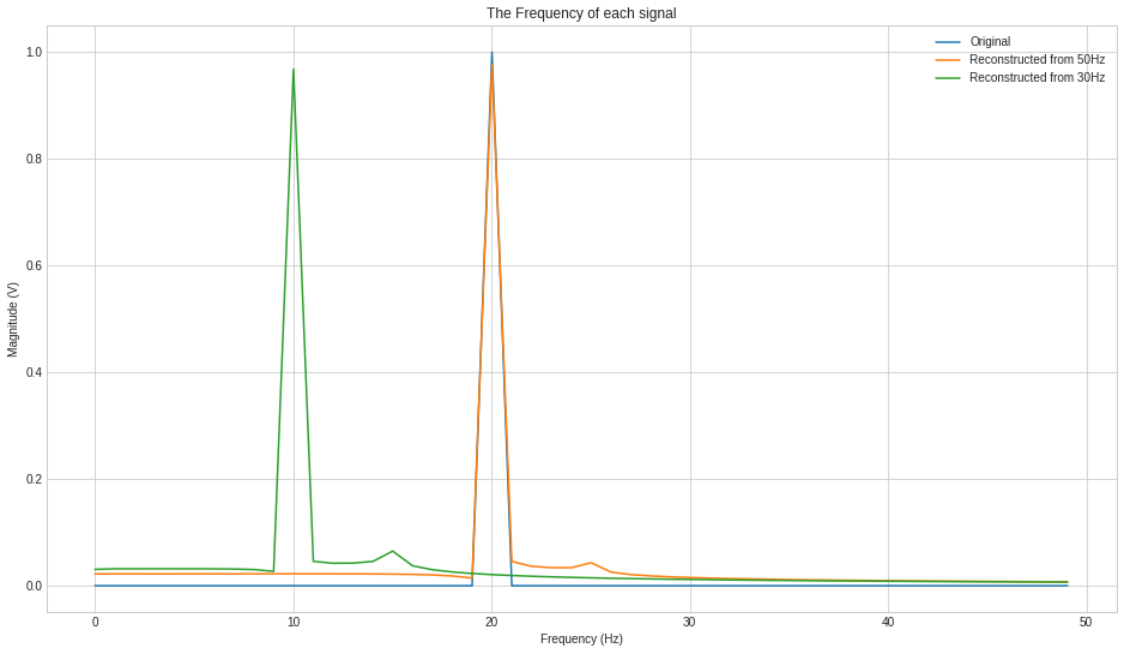
[7]:
```
# 6. Notice how the first reconstructed cosine has the same frequency as the original while the second has a different frequency.
# Clearly in the second reconstructed signal is not a perfect reconstruction.
# The new, folded frequency is 10 Hz. This is because the original frequency of 20 Hz is being folded over 30/2 = 15 Hz.

m1,f1 = calFFT(cos_20,window = None, normf=False, fs=1000)
m2,f2 = calFFT(inter_cos_50,window = None, normf=False,fs=1000)
m3,f3 = calFFT(inter_cos_30,window = None, normf=False, fs=1000)
print('Frequency of Original 20Hz cosine:',f1[np.argmax(m1)],'Hz')
print('Frequency of Reconstructed 20Hz cosine from 50Hz sampled data:',f2[np.argmax(m2)],'Hz')
print('Frequency of Reconstructed 20Hz cosine from 30Hz sampled data:',f3[np.argmax(m3)],'Hz')
```

```
Frequency of Original 20Hz cosine: 20.02002002002002 Hz
Frequency of Reconstructed 20Hz cosine from 50Hz sampled data: 20.02002002002002
Hz
Frequency of Reconstructed 20Hz cosine from 30Hz sampled data: 10.01001001001001
Hz
```

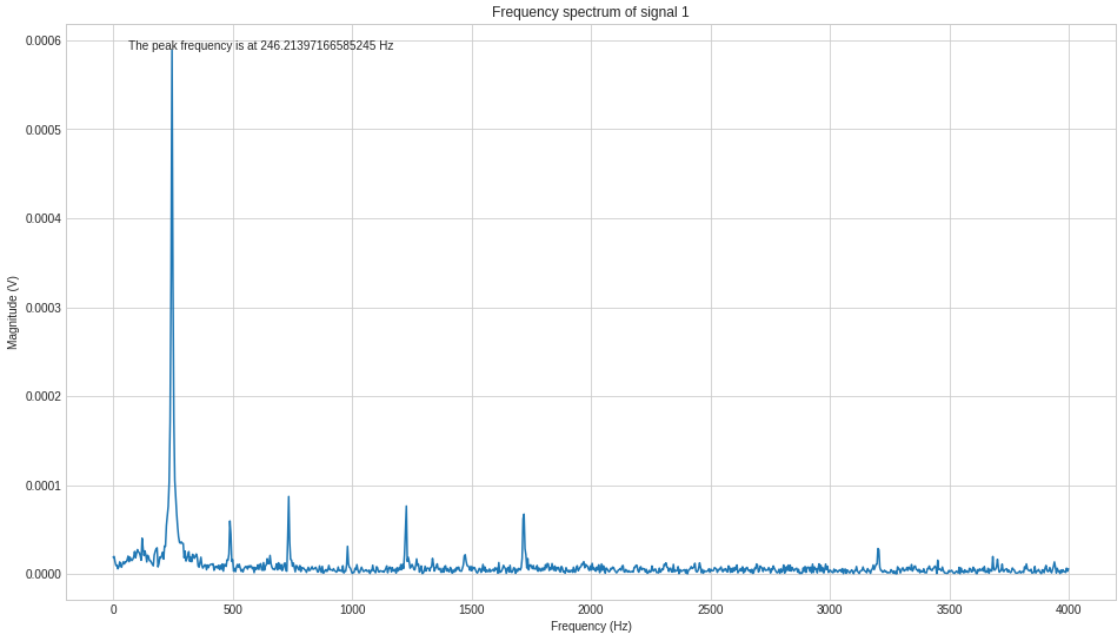
[8]:
```
# 7. Calculate the frequency spectrum of original and two reconstructed signals using am spectrum function and plot it using am plot function.
# Verify the frequencies of signals and the aliasing.

fig, ax = plt.subplots(1, figsize=(16,9))
```

```
ax.plot(f1[:50],m1[:50], label='Original')
ax.plot(f2[:50],m2[:50], label='Reconstructed from 50Hz')
ax.plot(f3[:50],m3[:50], label='Reconstructed from 30Hz')
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel('Magnitude (V)')
ax.set_title('The Frequency of each signal')
ax.legend()
```

[8]: <matplotlib.legend.Legend at 0x7f024c74a880>

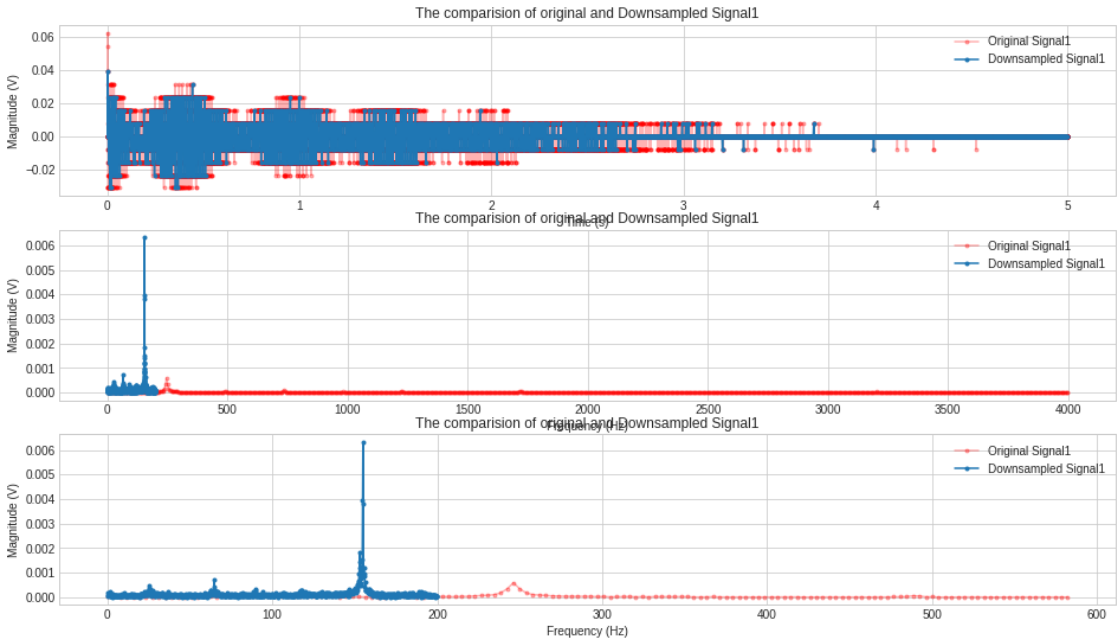


[9]:
```
# 8. Sampling Music - Load the given music sample and look at its spectrum.

fs = 8000 # Hz
duration = 5 # s
# signal1
signal1 = np.loadtxt('signal1.txt', dtype='double')
m_signal1,f_signal1 = calFFT(signal1,normf=False,fs=fs)
fig, ax = plt.subplots(1, figsize=(16,9))

ax.plot(f_signal1, m_signal1)
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel('Magnitude (V)')
ax.set_title(f'Frequency spectrum of signal 1')
ax.text(np.argmax(m_signal1), np.max(m_signal1), f"The peak frequency is at␣
 ↪{f_signal1[np.argmax(m_signal1)]} Hz")
plt.show()
```



[10]:
```
# 9. The spectrum extends up to 16 kHz. This is to be expected since the␣
 ↪sampling rate fs of the original signal is 32kHz.
# If this signal is sampled with anything less than 32 kHz there will be␣
 ↪aliasing.
# Use one eighth of the original sampling frequency that is 32/8=4 kHz.
# This means that frequencies only up to 2kHz will be represented.

fs = 8000 #Hz
# Alias range is 4001 - 8000 Hz

new_fs = 400 #Hz ==> 1000
# New Alias range is 501 - 1000

signal1_1000 = resample_by_interpolation(signal1, fs, new_fs)
t_1000 = np.linspace(0, 5, new_fs*5, endpoint=False)

fig,ax = plt.subplots(3,figsize=(16,9))
ax[0].plot(np.linspace(0,5,8000*5), signal1, 'r.-', alpha=0.3, label="Original␣
 ↪Signal1")
ax[0].plot(t_1000,signal1_1000, '.-', label="Downsampled Signal1")
ax[0].legend()
ax[0].set_xlabel('Time (s)')
ax[0].set_ylabel('Magnitude (V)')
ax[0].set_title('The comparision of original and Downsampled Signal1')

m_signal1_1000, f_signal1_1000 = calFFT(signal1_1000, normf=False, fs=new_fs)
```

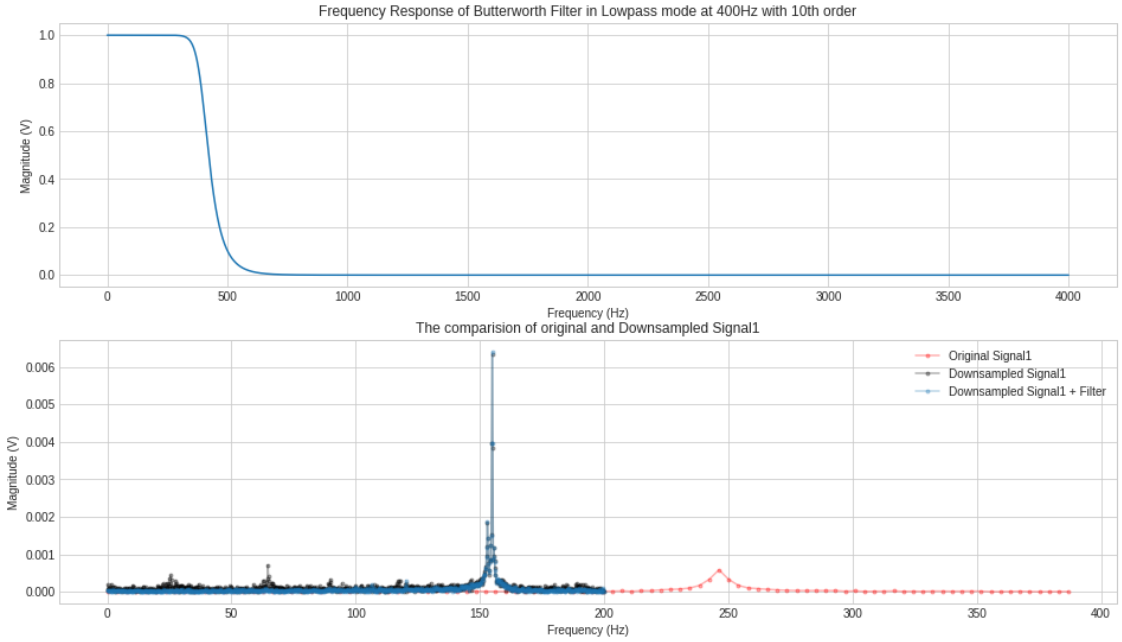
```
ax[1].plot(f_signal1, m_signal1, 'r.-', alpha=0.3, label="Original Signal1")
ax[1].plot(f_signal1_1000,m_signal1_1000, '.-', label="Downsampled Signal1")
ax[1].legend()
ax[1].set_xlabel('Frequency (Hz)')
ax[1].set_ylabel('Magnitude (V)')
ax[1].set_title('The comparision of original and Downsampled Signal1')


ax[2].plot(f_signal1[:150], m_signal1[:150], 'r.-', alpha=0.3, label="Original␣
 ↪Signal1")
ax[2].plot(f_signal1_1000,m_signal1_1000, '.-', label="Downsampled Signal1")
ax[2].legend()
ax[2].set_xlabel('Frequency (Hz)')
ax[2].set_ylabel('Magnitude (V)')
ax[2].set_title('The comparision of original and Downsampled Signal1')

plt.show()
```



[11]:
```
# 10. The signal now sounds dull, it lost its brightness since the high␣
 ↪frequencies are gone.
# Actually those high frequencies are folded because of aliasing thus creating␣
 ↪distortion.
# Now repeat the procedure but by using a low pass anti aliasing filter.
```

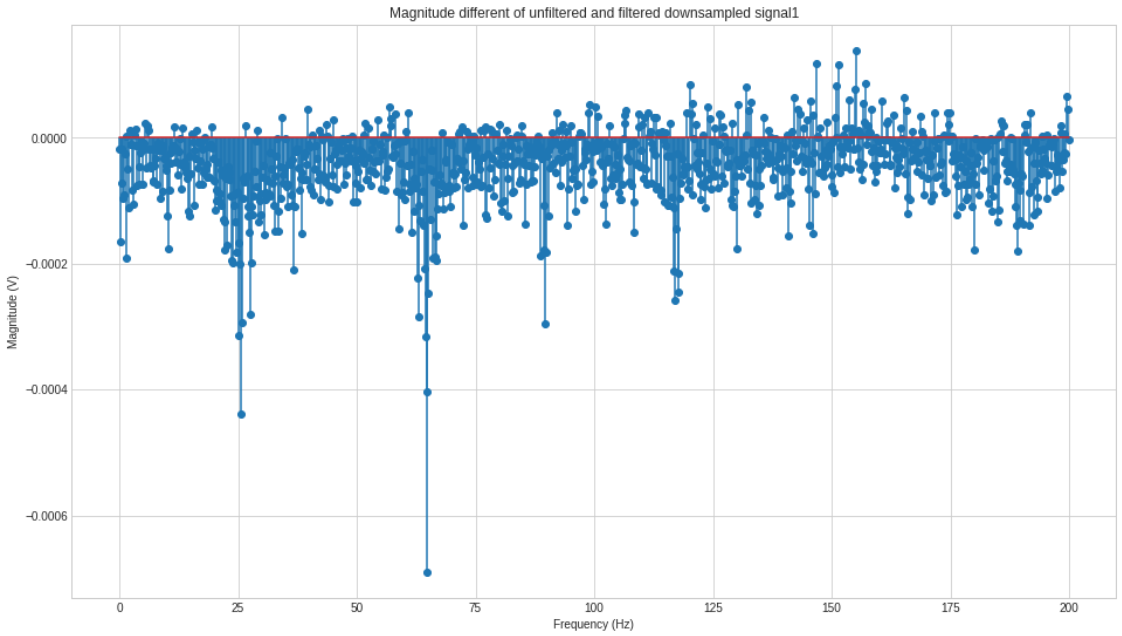
```
# Copy Filter Creator from Lab4
filter_bw = 0
filter_c1 = 1
filter_c2 = 2
filter_el = 3
filter_name = ['Butterworth', 'Chebyshev 1', 'Chebyshev 2', 'Ellipticol']
def getFilter(filter=0,order=2,lowcut=-1,highcut=-1,fs=500, rp=2, rs=20):
    b,a = None, None
    cut_off = None
    btype = None
    if(lowcut != -1 and highcut != -1):
        cut_off = [lowcut,highcut]
        btype = 'bandpass'
    elif(lowcut != -1):
        cut_off = lowcut
        btype = 'highpass'
    elif(highcut != -1):
        cut_off = highcut
        btype = 'lowpass'
    else:
        raise ValueError(f"Both lowcut and highcut cannot be -1 at the same␣
↪time")

    if(filter == 0):
        # scipy.signal.butter(N, Wn, btype='low', analog=False, output='ba',␣
↪fs=None)
        b,a = signal.butter(N=order,Wn=cut_off,btype=btype,output='ba',fs=fs)
    elif(filter == 1):
        # scipy.signal.cheby1(N, rp, Wn, btype='low', analog=False,␣
↪output='ba', fs=None)
        b,a = signal.cheby1(N=order,Wn=cut_off,btype=btype,output='ba',fs=fs,␣
↪rp=rp)
    elif(filter == 2):
        # scipy.signal.cheby2(N, rs, Wn, btype='low', analog=False,␣
↪output='ba', fs=None)
        b,a = signal.cheby2(N=order,Wn=cut_off,btype=btype,output='ba',fs=fs,␣
↪rs=rs)
    elif(filter == 3):
        # scipy.signal.ellip(N, rp, rs, Wn, btype='low', analog=False,␣
↪output='ba', fs=None)
        b,a = signal.ellip(N=order,Wn=cut_off,btype=btype,output='ba',fs=fs,␣
↪rp=rp,rs=rs)
    else:
        raise ValueError(f"No filter type {filter}")
    return b,a
```

```
def getFreqResponse(b,a,fs):
    w,n = signal.freqz(b,a,worN=(fs//2))
    return w/np.pi,abs(n)

b,a = getFilter(filter_bw, order=10, highcut=new_fs, fs=fs)
w,n = getFreqResponse(b,a,fs=fs)
fig,ax = plt.subplots(2, figsize=(16,9))
ax[0].plot(w*4000,n)
ax[0].set_xlabel("Frequency (Hz)")
ax[0].set_ylabel("Magnitude (V)")
ax[0].set_title(f"Frequency Response of Butterworth Filter in Lowpass mode at␣
 ↪{new_fs}Hz with 10th order")

filtered_signal1 = signal.lfilter(b,a,signal1)

filtered_signal1_1000 = resample_by_interpolation(filtered_signal1, fs, new_fs)
m_filtered_signal1_1000, f_filtered_signal1_1000 =␣
 ↪calFFT(filtered_signal1_1000, normf=False, fs=new_fs)


ax[1].plot(f_signal1[:100], m_signal1[:100], 'r.-', alpha=0.3, label="Original␣
 ↪Signal1")
ax[1].plot(f_signal1_1000, m_signal1_1000, 'k.-', alpha=0.3, label="Downsampled␣
 ↪Signal1")
ax[1].plot(f_filtered_signal1_1000,m_filtered_signal1_1000, '.-',alpha=0.3,␣
 ↪label="Downsampled Signal1 + Filter")
ax[1].legend()
ax[1].set_xlabel('Frequency (Hz)')
ax[1].set_ylabel('Magnitude (V)')
ax[1].set_title('The comparision of original and Downsampled Signal1')
plt.show()
```



[12]:
```
# 11. Compare the signals md1 and md2 and listen to the difference with the␣
 ↪following simple command: soundsc(md1 - md2,newfs)

fig,ax = plt.subplots(1, figsize=(16,9))
ax.stem(f_signal1_1000, m_filtered_signal1_1000 - m_signal1_1000)
ax.set_xlabel("Frequency (Hz)")
ax.set_ylabel("Magnitude (V)")
ax.set_title(f"Magnitude different of unfiltered and filtered downsampled␣
 ↪signal1")
```

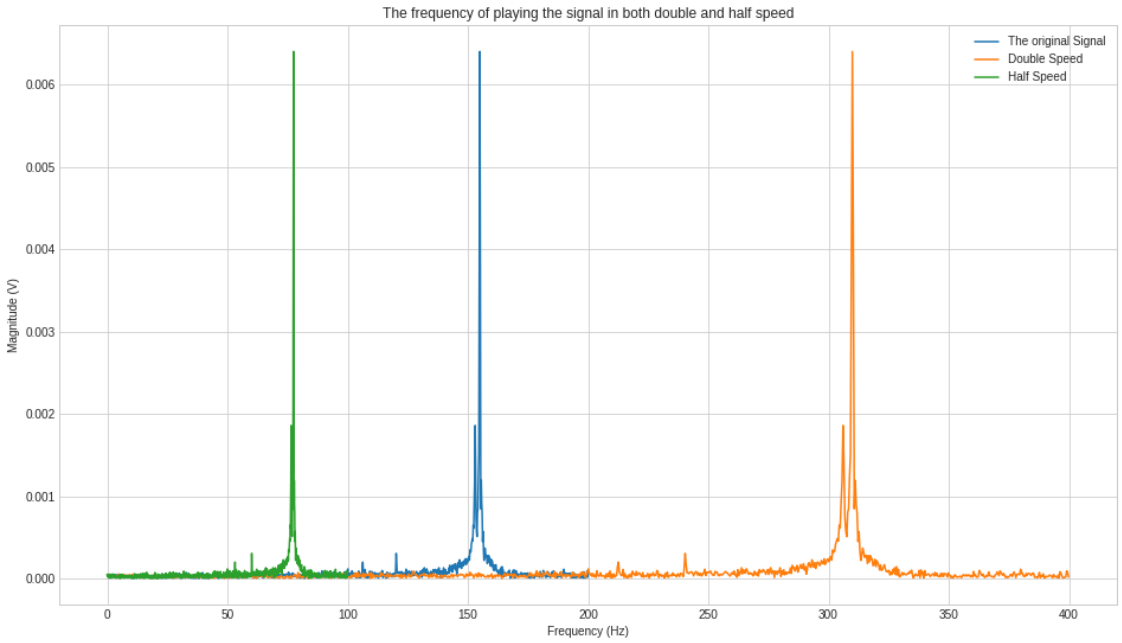
[12]:
```
Text(0.5, 1.0, 'Magnitude different of unfiltered and filtered downsampled
signal1')
```



[13]:
```
# 12. By comparison of the three spectra it can be seen that in the case of no␣
 ↪anti aliasing filter
# there is more energy on the 1-2 kHz band compared to the case with anti␣
 ↪aliasing filter.
# This is because the anti aliasing filter prevents frequencies from being␣
 ↪folded over.

# This is done in the above figure
# In the case of this signal1, there are more frequency around 0 - 100 Hz in␣
 ↪the unfiltered downsampled signal
```

[14]:
```
# 13. Try to change the playback rate of the original sound using the command
# - soundsc(m,fs/2) for half the speed
# - soundsc(m,fs*2) for double speed.
# This is a practical application of the scaling property of the Fourier␣
 ↪transform.

fig,ax = plt.subplots(1, figsize=(16,9))
ax.plot(f_signal1_1000, m_filtered_signal1_1000, label="The original Signal")
ax.plot(f_signal1_1000 * 2, m_filtered_signal1_1000, label="Double Speed")
ax.plot(f_signal1_1000 / 2, m_filtered_signal1_1000, label="Half Speed")
ax.legend()
ax.set_xlabel("Frequency (Hz)")
ax.set_ylabel("Magnitude (V)")
ax.set_title(f"The frequency of playing the signal in both double and half␣
 ↪speed")
```

[14]: `Text(0.5, 1.0, 'The frequency of playing the signal in both double and half speed')`



2 Discussion

2.1 1. Explain all the Matlab functions used in this experiment.

signal.unit_impulse: to create the impulse train

resample_by_interpolation: to resample the signal. There is a resample function in scipy but for some reason, I does not work when I try to downsample the signal.

sinc_interp: Is a function written follow the interpsinc in matLab. It is use for reconstruct the signal.

2.2 2. Following the procedure described in the first part of the lab check for aliasing and calculate the frequency of the reconstructed cosines in the following cases:

2.2.1 (a) Cosine: 30Hz, Sampling: 50Hz

Freq range = 25 Hz

Record Frequency = 25 - (30 % 25)

- 20 Hz

2.2.2 (b) Cosine: 40Hz, Sampling: 15Hz

Freq range = 7.5 Hz

Record Frequency = 7.5 - (40 % 7.5)

- 5 Hz

2.2.3 (c) Cosine: 10Hz, Sampling: 50Hz

Freq range = 25 Hz

Record Frequency = 25 - (10 % 25)

- 10 Hz

2.2.4 (d) Cosine: 20Hz, Sampling: 40Hz

Freq range = 20 Hz

Record Frequency = 20 - (20 % 20)

- 20 Hz or 0 Hz depends on the signal. If is a sin wave then it will be 0Hz.

[ ]: