DT2470 Lab 01: Teh Signal Processings

by Bob L. T. Sturm

In this first lab you will practice some fundamental concepts of signal processing. You will analyse a chosen sampled sound in the time-, frequency-, and time-frequency domains. You will write something intelligent about your analysis, observing things like periodicity, frequency content, harmonicity, etc. You will also learn to extract low-level features from audio and music signals. In the next lab, you will use these features for some machine learning madness.

The lab report you submit should be a testament to your intelligence, as well as a reflection of your willingness to be a part of this module. You are free to use whatever software you want, e.g., python, MATLAB, Processing, C++, etc. But I give tips below in python. Here's some helpful links as well:

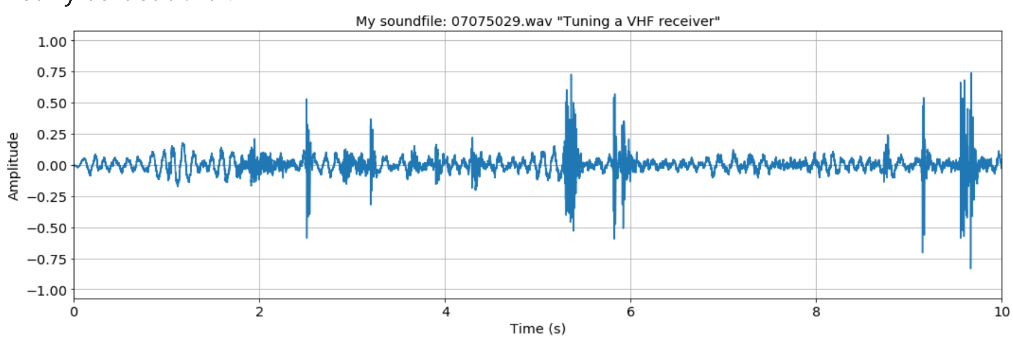
- Numpy API
- Scikit-learn API
- MatPlotlib API
- Numpy Cheat Sheet
- Pydub API

I also include some images so you can confirm whether you are on the right track, or just to have a brief pause to laugh at how far your answer is from being correct.

Names: Sergi Andreu and Carsten van de Kamp

Part 1: Basics

1. Choose an audio file to work with from http://bbcsfx.acropolis.org.uk. Download it, load it using pydub (see pydub.AudioSegment), and plot a portion of the waveform with the appropriate axes labeled "Amplitude" and "Time (s)". The time axis **must be** in seconds. (Use the sample rate of your soundfile to find that.) If your audio file has more than one channel, just look at one channel.

> Below is the first 10 seconds of my selected audio waveform. Yours should appear nearly as beautiful.
>
> 
>

In [1]:
```
# Downloading and unzipping our audio file

import os
```

```python
import wget
from zipfile import ZipFile

snd_dir = 'tmp/snd'
if not os.path.exists(snd_dir):
    os.makedirs(snd_dir)

url = 'https://sound-effects-media.bbcrewind.co.uk/zip/07042033.wav.zip?download'
filename = '07042033.wav'

# Download
file = wget.download(url, snd_dir)

# Unzip
zip = ZipFile(file)
zip.extractall(snd_dir)
zip.close
```

```
34% [.........................                                                ] 25
80480 / 7381337100%
[...........................................................................] 7381337
/ 7381337
```

Out[1]:

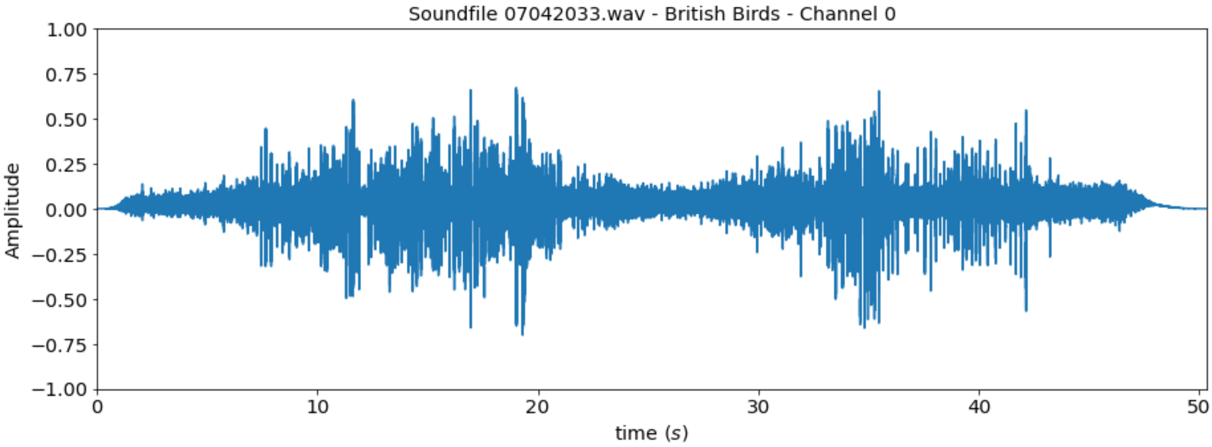
```
<bound method ZipFile.close of <zipfile.ZipFile filename='tmp/snd/07042033.wav (1).z
ip' mode='r'>>
```

In [2]:

```python
# Now we plot the entire audio file
import pydub
import matplotlib.pyplot as plt
import numpy as np

# The following makes the plot look nice
params = {'legend.fontsize': 'x-large',
          'figure.figsize': (15, 5),
          'axes.labelsize': 'x-large',
          'axes.titlesize':'x-large',
          'xtick.labelsize':'x-large',
          'ytick.labelsize':'x-large'}
plt.rcParams.update(params)

# add your code below

#Load the sound file and some properties
sound = pydub.AudioSegment.from_file(snd_dir + '/' + filename, format="wav", duratio
                                     channels = 2, frame_rate = 44100, sample_width

sound_mono = sound.split_to_mono()
samples = [[],[]]
samples[0] = sound_mono[0].get_array_of_samples()
samples[1] = sound_mono[1].get_array_of_samples()

# Choose either of the 2 channels that are in the audio file
ind = 0

sample_rate = sound_mono[ind].frame_rate
nr_channels = sound_mono[ind].channels
duration    = sound_mono[ind].duration_seconds

# Normalizing both channels
max_possible_amplitude_ch1 = sound_mono[0].max_possible_amplitude
max_possible_amplitude_ch2 = sound_mono[1].max_possible_amplitude
samples[0] = np.array(samples[0]) / max_possible_amplitude_ch1
samples[1] = np.array(samples[1]) / max_possible_amplitude_ch2
```

```python
#Plotting
time_axis = np.arange(0, duration, 1/sample_rate)

fig, ax = plt.subplots()
ax.plot(time_axis, samples[ind])
ax.set_xlim((0,duration))
ax.set_ylim((-1,1))
ax.set_xlabel(r"time ($s$)")
ax.set_ylabel(r"Amplitude")
ax.set_title(f"Soundfile {filename} - British Birds - Channel {ind}")
```

Out[2]: Text(0.5, 1.0, 'Soundfile 07042033.wav - British Birds - Channel 0')



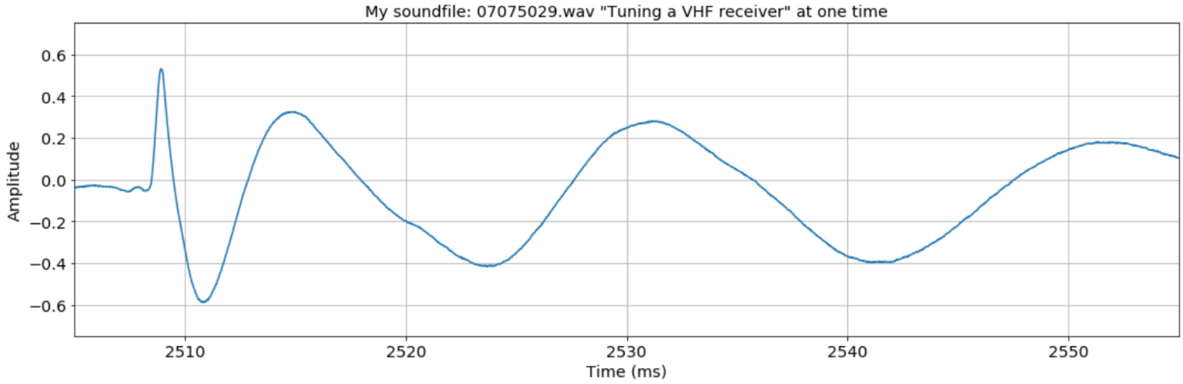
In [3]:
```python
import IPython

IPython.display.Audio(samples[ind], rate=sample_rate)
```

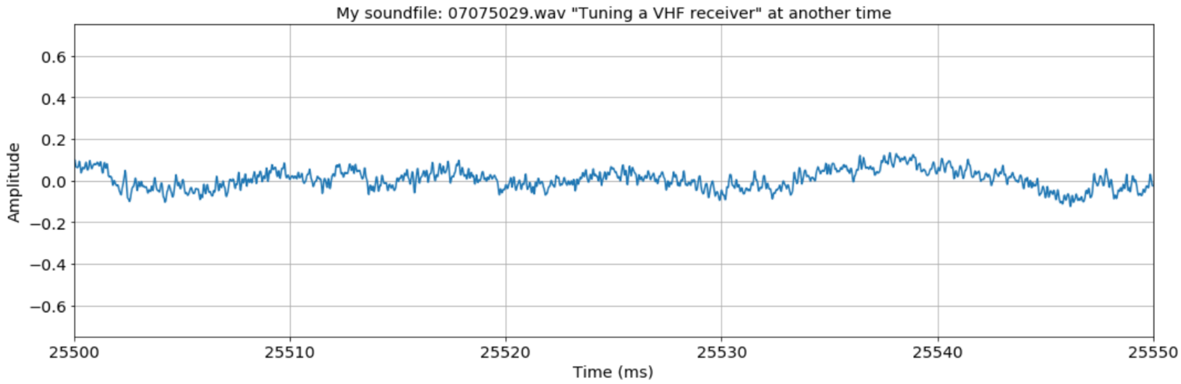
Out[3]:

0:00 / 0:50

1. With the audio file you have chosen, zoom into two different 100 ms portions that have audio data and plot them.

> Below is what mine looks like. We can see the sound appears quite different at both times. At about 2500 ms we see a sudden rise that decays and oscillates. At about 26 s we see a noisy waveform that has a small amplitude.

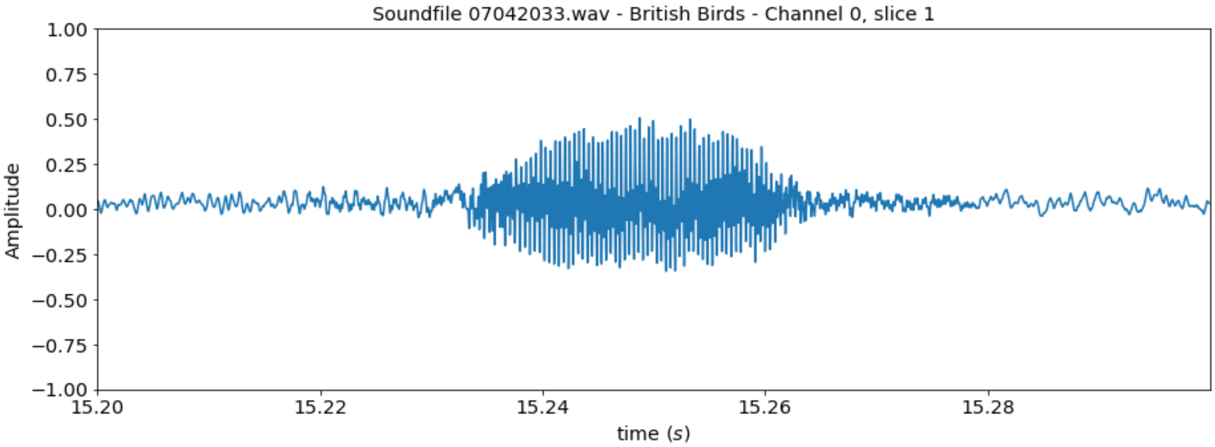


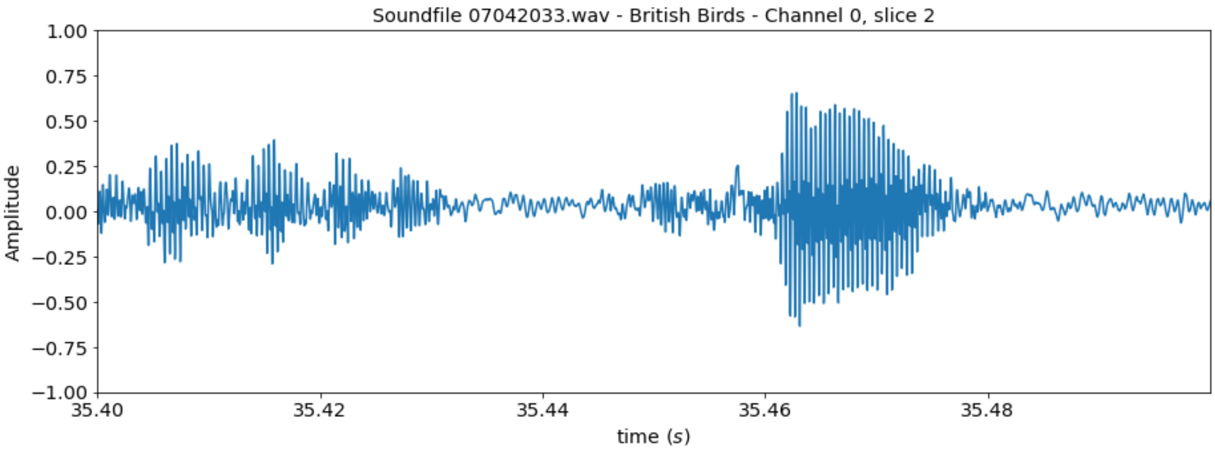


In [4]:

```
# add your code below

# Choose two interesting time intervals of length 0.1s
time_interval = 0.1
time_interval1 = [15.2, 15.3]
time_interval2 = [35.4, 35.5]

# Slicing the time axis and audio samples
num_samples1 = int((time_interval1[1] - time_interval1[0]) * sample_rate)
num_samples2 = int((time_interval2[1] - time_interval2[0]) * sample_rate)

slice1_first_idx = int(time_interval1[0]*sample_rate)
slice2_first_idx = int(time_interval2[0]*sample_rate)

time_axis1 = time_axis[slice1_first_idx : slice1_first_idx + num_samples1]
time_axis2 = time_axis[slice2_first_idx : slice2_first_idx + num_samples2]

samples_slice1 = samples[ind][slice1_first_idx : slice1_first_idx + num_samples1]
samples_slice2 = samples[ind][slice2_first_idx : slice2_first_idx + num_samples2]

# Plotting
fig1, ax1 = plt.subplots()
fig2, ax2 = plt.subplots()
ax1.plot(time_axis1, samples_slice1)
ax2.plot(time_axis2, samples_slice2)

ax1.set_xlim((time_axis1[0],time_axis1[-1]))
ax1.set_ylim((-1,1))
ax1.set_xlabel(r"time ($s$)")
ax1.set_ylabel(r"Amplitude")
ax1.set_title(f"Soundfile {filename} - British Birds - Channel {ind}, slice 1")

ax2.set_xlim((time_axis2[0],time_axis2[-1]))
ax2.set_ylim((-1,1))
ax2.set_xlabel(r"time ($s$)")
ax2.set_ylabel(r"Amplitude")
ax2.set_title(f"Soundfile {filename} - British Birds - Channel {ind}, slice 2")
```

Out[4]: Text(0.5, 1.0, 'Soundfile 07042033.wav - British Birds - Channel 0, slice 2')





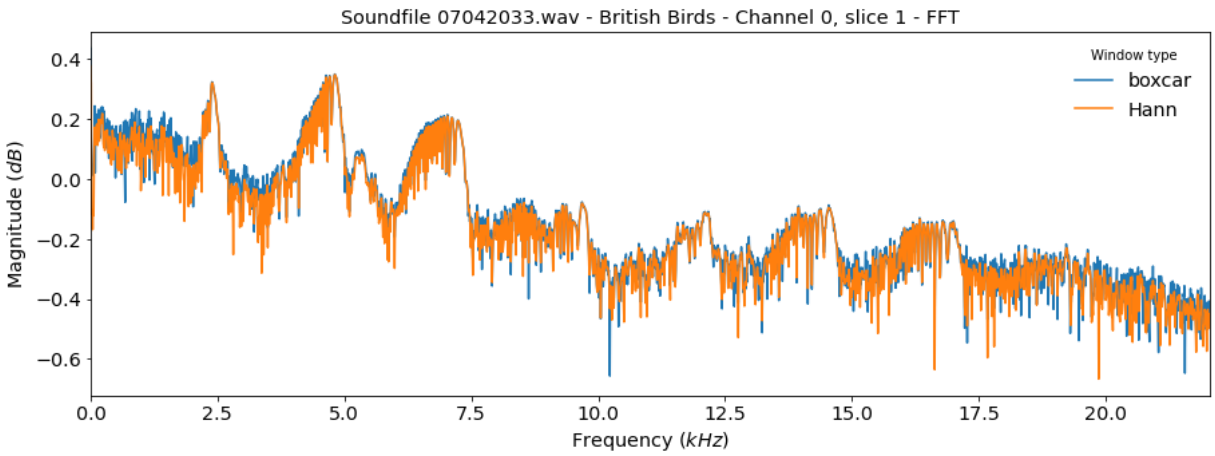
1. For each of the segments you looked at above, window them, and compute their Fourier transforms. Plot their dB magnitude spectra. Appropriately label your axes with "Magnitude (dB)" and "Frequency (kHz)". The frequency axis **must be** in kiloHertz, and limited to 0 to the Nyquist frequency (half the sampling rate). Window the audio with 1) boxcar, or 2) Hann. (This means you will have create four plots in total , or two plots with two lines each.)

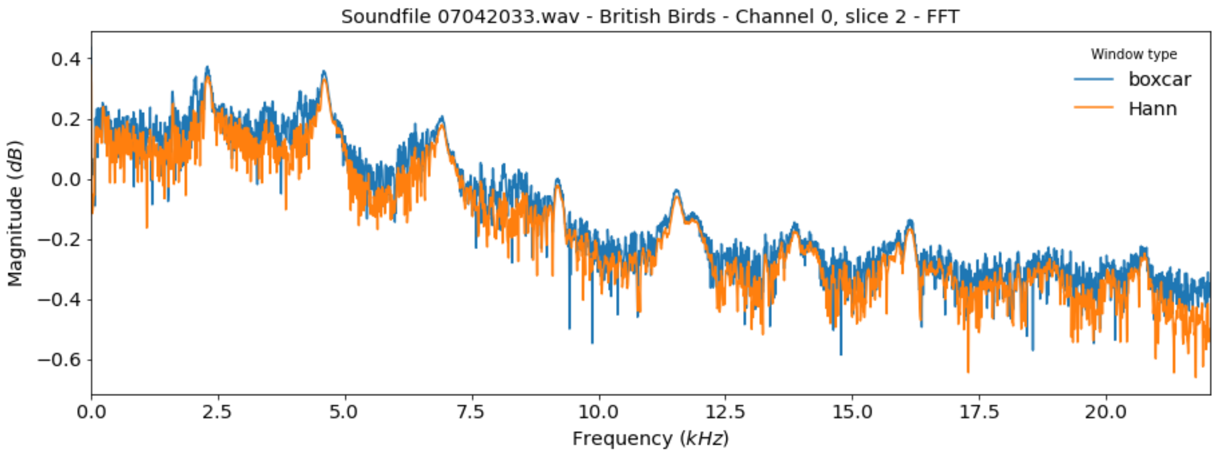
In [5]:

```
# add your code below
from scipy.signal import get_window

def FFT_window_segment(sample_slice, window_type):
    """
    Computes the FFT magnitude and phase of an audio sample with a certain window ty
    The window size is equal to the sample size
    """
    window = get_window(window_type, len(sample_slice))
    windowed_sample = window*sample_slice
    FFT_windowed_sample = np.fft.fft(windowed_sample)
    mag = np.abs(FFT_windowed_sample)
    phase = np.angle(FFT_windowed_sample)
    return mag, phase

Nyquist = sample_rate/2
Nyquist_idx = int(Nyquist * time_interval)                # index of the Nyquist
frequencies = np.arange(0, Nyquist, 1/time_interval) /1000    # (kHz)  # the resolut

# Compute the FFT for the different segments with both boxcar and Hann window
mag1_b, phase1_b = FFT_window_segment(samples_slice1, 'boxcar')
mag1_h, phase1_h = FFT_window_segment(samples_slice1, 'hann')

mag2_b, phase2_b = FFT_window_segment(samples_slice2, 'boxcar')
mag2_h, phase2_h = FFT_window_segment(samples_slice2, 'hann')
```

```python
# Plotting
fig1, ax1 = plt.subplots()
ax1.plot(frequencies, np.log10(mag1_b[:Nyquist_idx])/5, label='boxcar')
ax1.plot(frequencies, np.log10(mag1_h[:Nyquist_idx])/5, label='Hann')
ax1.set_xlim((0,Nyquist/1000))
ax1.set_title(f"Soundfile {filename} - British Birds - Channel {ind}, slice 1 - FFT"
ax1.set_xlabel(r'Frequency $(kHz)$')
ax1.set_ylabel(r'Magnitude $(dB)$')
ax1.legend(title='Window type', frameon=False)

fig2, ax2 = plt.subplots()
ax2.plot(frequencies, np.log10(mag2_b[:Nyquist_idx])/5, label='boxcar')
ax2.plot(frequencies, np.log10(mag2_h[:Nyquist_idx])/5, label='Hann')
ax2.set_xlim((0,Nyquist/1000))
ax2.set_title(f"Soundfile {filename} - British Birds - Channel {ind}, slice 2 - FFT"
ax2.set_xlabel(r'Frequency $(kHz)$')
ax2.set_ylabel(r'Magnitude $(dB)$')
ax2.legend(title='Window type', frameon=False)
```

Out[5]: <matplotlib.legend.Legend at 0x22f163fa220>





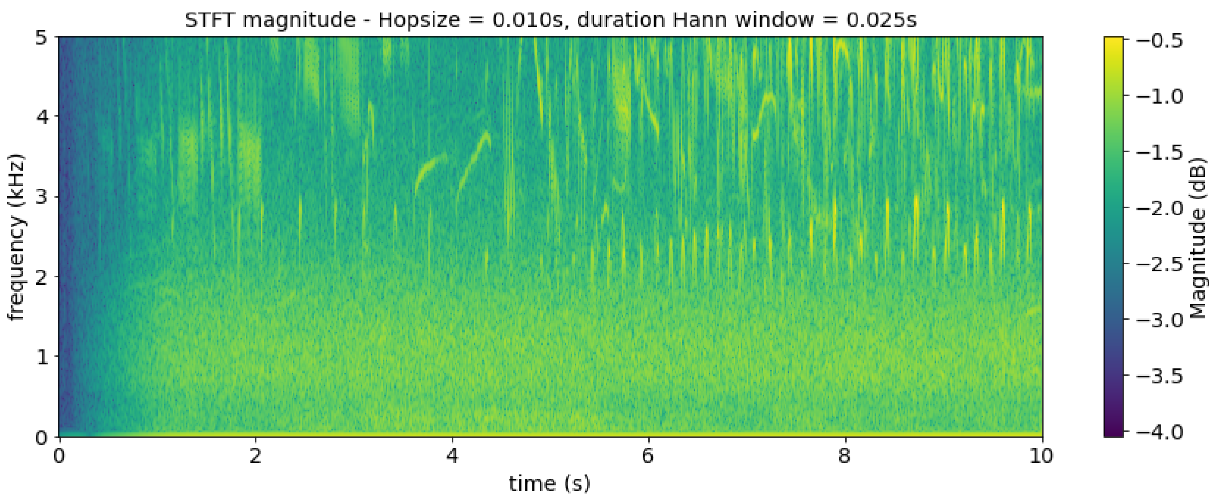
1. For the first 10 seconds of your audio file, compute and plot its dB magnitude short-time Fourier transform using a Hann window of duration 25 ms with a window hopsize of 10 ms, and an FFT size of 8192 samples. Do the same using a Hann window of duration 100 ms with a window hopsize of 10 ms. Appropriately label your axes with "Frequency (kHz)" and "Time (s)". The frequency axis must be in kiloHertz, and limited to 0 to 5 kHz. The time axis must be in seconds. Choose a colormap that you feel describes your personality (https://matplotlib.org/3.1.1/tutorials/colors/colormaps.html). See scipy.signal for help.

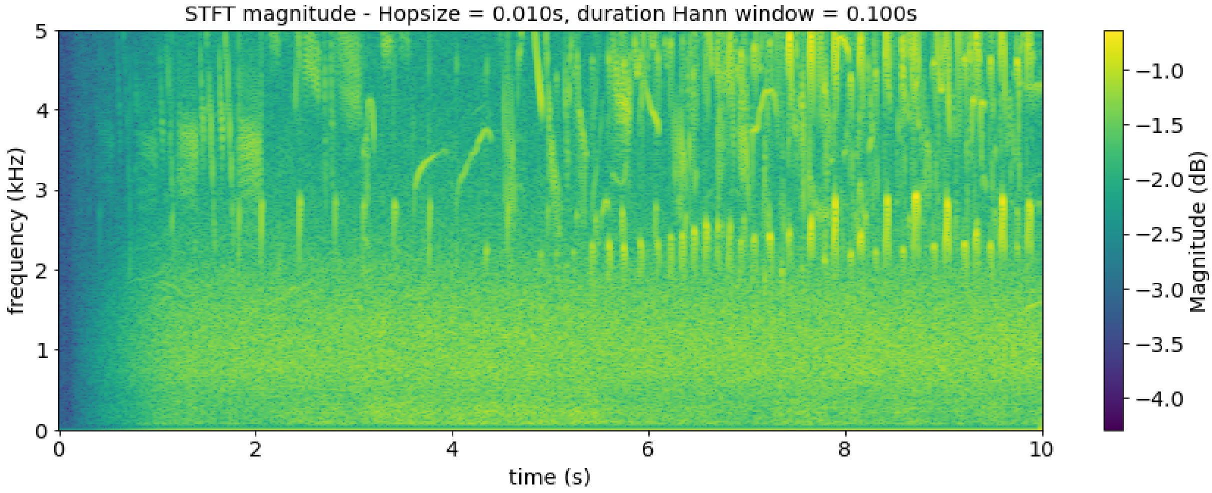
In [6]:

```python
# add your code below
from scipy.signal import stft

# Slice first 10s
num_samples_in_10s = int(10 * sample_rate)
start = int(num_samples_in_10s * 0)
end   = int(num_samples_in_10s * 1)
time_axis10s = time_axis[start : end]
samples_slice10s = samples[ind][start : end]

nperseg1 = 0.025 * sample_rate       # Length of each segment = window duration
hopsize1 = 0.010 * sample_rate
noverlap1 = nperseg1 - hopsize1      # Number of points to overlap between segments,

nperseg2 = 0.100 * sample_rate       # Length of each segment = window duration
hopsize2 = 0.010 * sample_rate
noverlap2 = nperseg2 - hopsize2      # Number of points to overlap between segments,

# Compute short-time Fourier transform
f1, t1, Zxx1 = stft(samples_slice10s, fs=sample_rate, window='hann', nperseg=nperseg
f2, t2, Zxx2 = stft(samples_slice10s, fs=sample_rate, window='hann', nperseg=nperseg

# Get the right units
f1_kHz = f1/1000
f2_kHz = f2/1000

dB_mag1 = np.log(np.abs(Zxx1))/5
dB_mag2 = np.log(np.abs(Zxx2))/5

#Plotting
fig1, ax1 = plt.subplots()
im1 = ax1.pcolormesh(t1, f1_kHz, dB_mag1, shading='nearest', cmap='viridis')
ax1.set_xlabel('time (s)')
ax1.set_ylabel('frequency (kHz)')
ax1.set_ylim((0,5))
ax1.set_title(f'STFT magnitude - Hopsize = 0.010s, duration Hann window = 0.025s')
fig1.colorbar(im1, label = 'Magnitude (dB)')

fig2, ax2 = plt.subplots()
im2 = ax2.pcolormesh(t2, f2_kHz, dB_mag2, shading='nearest', cmap='viridis')
ax2.set_xlabel('time (s)')
ax2.set_ylabel('frequency (kHz)')
ax2.set_ylim((0,5))
ax2.set_title(f'STFT magnitude - Hopsize = 0.010s, duration Hann window = 0.100s')
fig2.colorbar(im2, label = 'Magnitude (dB)')
```

Out[6]:

```
<matplotlib.colorbar.Colorbar at 0x22f169e89a0>
```





1. Describe some of the advantages and nackdelar of using short or long time windows for time-frequency analysis.

A long time window takes a bigger portion of sound into account and therefore there are probably more frequencies present in this window. This increases the frequency resolution of the STFT, but it reduces the time resolution. A very short time window has a low frequency resolution but has a higher time resolution making it easier to see at what time certain frequencies are present in the signal. Therefore, a trade-off has to be made between time and frequency resolution.

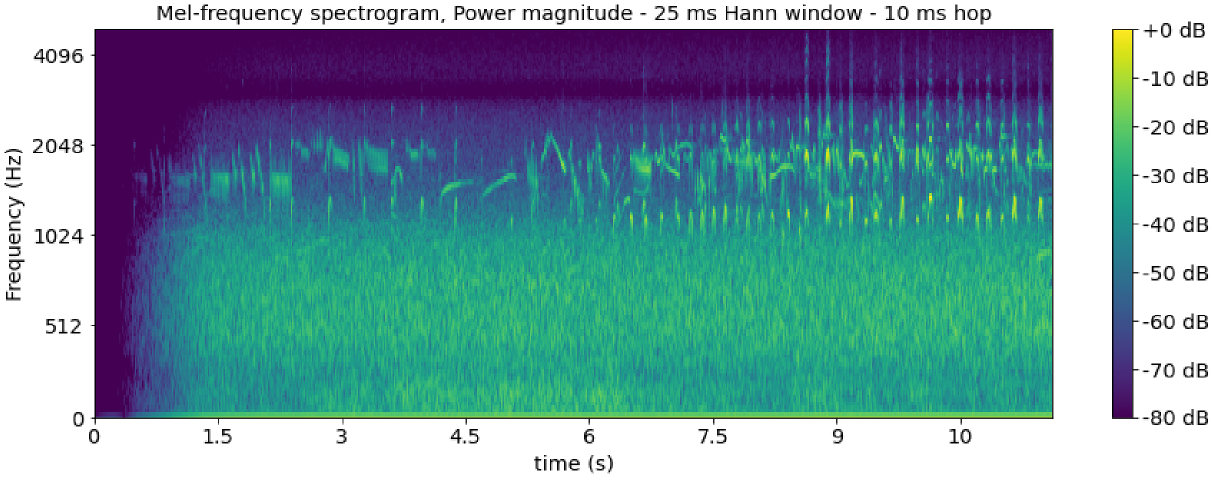
1. For the first 10 seconds of your audio file, use the librosa package to compute its Mel spectrogram using Hann windows of duration 25 ms with a window hopsize of 10 ms. Use 128 Mel bands and an FFT size of 8192 samples. Display the dB magnitude with reference to the max power observed, and limit your y-axis between 0 and 5 kHz. Use the same colormap as you used above. See https://github.com/librosa/librosa/blob/main/examples/LibROSA%20demo.ipynb for help.

In [7]:

```
import librosa
import librosa.display

hop_length = int(0.010 * sample_rate)
win_length = int(0.025 * sample_rate)


Mel_spectogram = librosa.feature.melspectrogram(y=samples_slice10s,
                                sr=sample_rate,
                                n_fft=8192,
                                hop_length=hop_length,
                                win_length=win_length,
                                window='hann',
                                n_mels = 128,
                                power=2.0)   # power=2 gives the power spectrum


# Compute power in dB relative to peak power
S_dB = librosa.power_to_db(Mel_spectogram, ref=np.max)

# Plotting
fig, ax = plt.subplots()
img = librosa.display.specshow(S_dB, x_axis='time', y_axis='mel', sr=sample_rate, fm
fig.colorbar(img, ax=ax, format='%+2.0f dB')
ax.set(title='Mel-frequency spectrogram, Power magnitude - 25 ms Hann window - 10 ms
```

```
        xlabel = 'time (s)',
        ylabel = 'Frequency (Hz)')
```

Out[7]:
```
[Text(0.5, 1.0, 'Mel-frequency spectrogram, Power magnitude - 25 ms Hann window - 10
ms hop'),
 Text(0.5, 0, 'time (s)'),
 Text(0, 0.5, 'Frequency (Hz)')]
```



Part 2: Extracting features

1. Write a function that will take in the samples of an audio file, a frame size in samples, a frame hop size in samples, and compute and return the number of waveform zero crossings in each frame. A waveform x[n] undergoes a zero crossing when sign(x[n]) and sign(x[n+1]) are different. You will have to slice x[n] into chunks of a specified size, and for each of those chunks, count the number of sign changes.

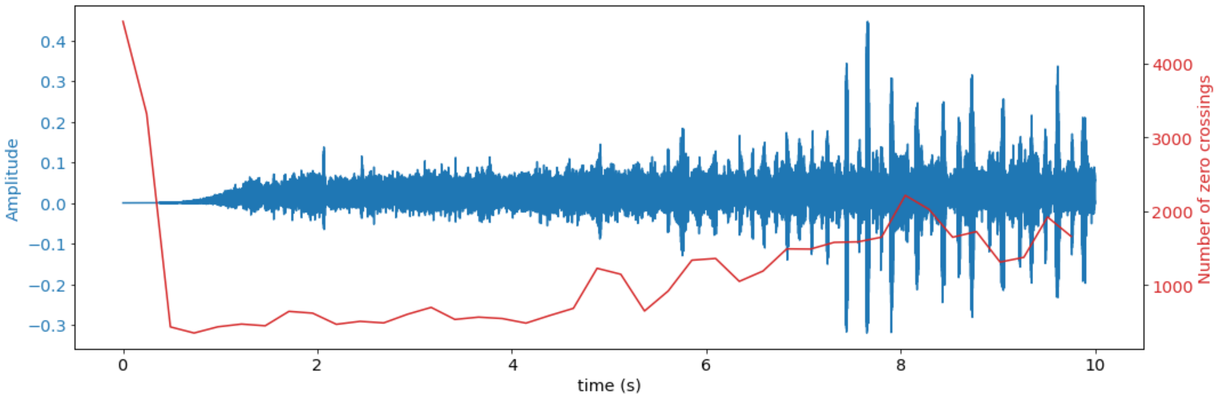
In [8]:
```
# add your code below

def count_zero_crossings(sample):
    """
    Count the number of zero crossings in a single sample
    """
    n_zero_crossings = np.shape(np.nonzero(np.diff(np.sign(sample))))[1]
    return n_zero_crossings

def slice_and_count_zero_crossings(sample, frame_size, hop_size):
    """
    Slice a sample according to frame_size and hop_size and find the number of
    zero crossings in each slice
    """
    L = np.shape(sample)[0]
    n_slices = int(( L - frame_size ) / hop_size)
    n_zeros = np.zeros((n_slices+1,))

    for i in range(n_slices + 1):
        n_zeros[i] = count_zero_crossings(sample[i*hop_size : i*hop_size + frame_siz

    return n_zeros
```

1. Using your function, compute zero crossings of 46 ms frames hopped 50% of that for the audio file you used in part 1. (Ignore any frames at the end of audio files that are less than

that length.) Plot the first 10 seconds of your time domain waveform, and plot the series of zero crossings you extracted.

In [9]:

```
# add your code below

window_time_frame = 0.46
frame_size = int(sample_rate * window_time_frame)
hop_size = int(frame_size / 2)

time = 10
time_size = int(time * sample_rate)

nzeros = slice_and_count_zero_crossings(samples[0], frame_size, hop_size)
nwindows = int( (time_size - frame_size) / hop_size)

nzeros = nzeros[:nwindows]
time_axis1 = (time / nwindows) * np.arange(0, nwindows, 1)

sample = samples[0][:time_size]
time_axis2 = 1 / (sample_rate) * np.arange(0, time_size, 1)

# Plotting
fig, ax1 = plt.subplots()
color = 'tab:blue'
ax1.set_xlabel('time (s)')
ax1.set_ylabel('Amplitude', color=color)  # we already handled the x-label with ax1
ax1.plot(time_axis2, sample, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()                         # instantiate a second set of axes that sh
color = 'tab:red'
ax2.set_ylabel('Number of zero crossings', color=color)
ax2.plot(time_axis1, nzeros, color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()                        # otherwise the right y-label is slightly
plt.show()
```

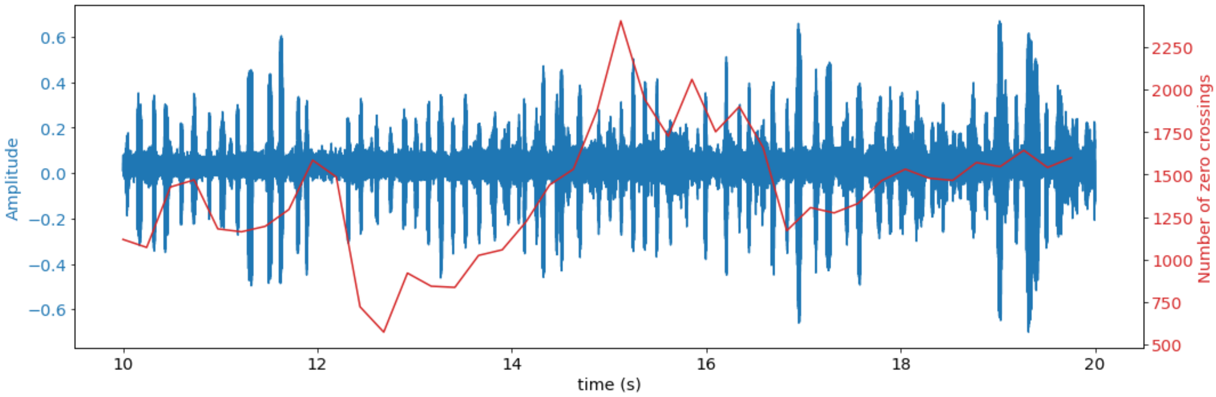


Because the first part of our sound file is not that interesting (noisy), we also plotted the part from t = 10 s to t = 20 s.

In [10]:

```
nzeros = slice_and_count_zero_crossings(samples[0], frame_size, hop_size)
nwindows = int( (time_size - frame_size) / hop_size)

nzeros = nzeros[nwindows:2*nwindows]
time_axis1 = (time / nwindows) * np.arange(nwindows, 2*nwindows, 1)

sample = samples[0][time_size:2*time_size]
time_axis2 = 1 / (sample_rate) * np.arange(time_size, 2*time_size, 1)
```

```
# Plotting
fig, ax1 = plt.subplots()
color = 'tab:blue'
ax1.set_xlabel('time (s)')
ax1.set_ylabel('Amplitude', color=color)  # we already handled the x-label with ax1
ax1.plot(time_axis2, sample, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()                          # instantiate a second set of axes that sh
color = 'tab:red'
ax2.set_ylabel('Number of zero crossings', color=color)
ax2.plot(time_axis1, nzeros, color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()                         # otherwise the right y-label is slightly
plt.show()
```



1. Write a function that will take in the samples of an audio file, a frame size in samples, a hop size in samples, and a sampling rate, and compute and return the spectral centroid of each frame. The spectral centroid of a rectangular window of audio $x[n]$ of length N (even) is defined as

$$R_{0.5}(x) = \frac{\sum_{k=0}^{N/2+1} \frac{F_s k}{N} |X[k]|}{\sum_{k=0}^{N/2+1} |X[k]|}$$

where $X[k]$ is the DFT of $x[n]$, and F_s is the sampling rate.

In [11]:

```
# add your code below

def spectral_centroid(sample, frame_size, hop_size, Fs):
    frame_size = int(frame_size)
    hop_size   = int(hop_size)
    number_of_frames = int( np.floor( ( len(sample) - frame_size ) / hop_size )  )

    start_idx = 0
    R = np.empty(number_of_frames)

    # Loop over every frame, then shift the frame index by the hop size
    for frame_nr in np.arange(0,number_of_frames):
        frame = sample[start_idx : start_idx+frame_size]

        X = np.fft.fft(frame)
        X_abs = np.abs(X)

        numerator = Fs / frame_size * np.sum(np.multiply(X_abs[0:int(frame_size/2)+2
```

```
        denominator = np.sum(X_abs[0:int(frame_size/2)+2])

        R[frame_nr] = numerator / denominator

        start_idx += hop_size

    return R
```

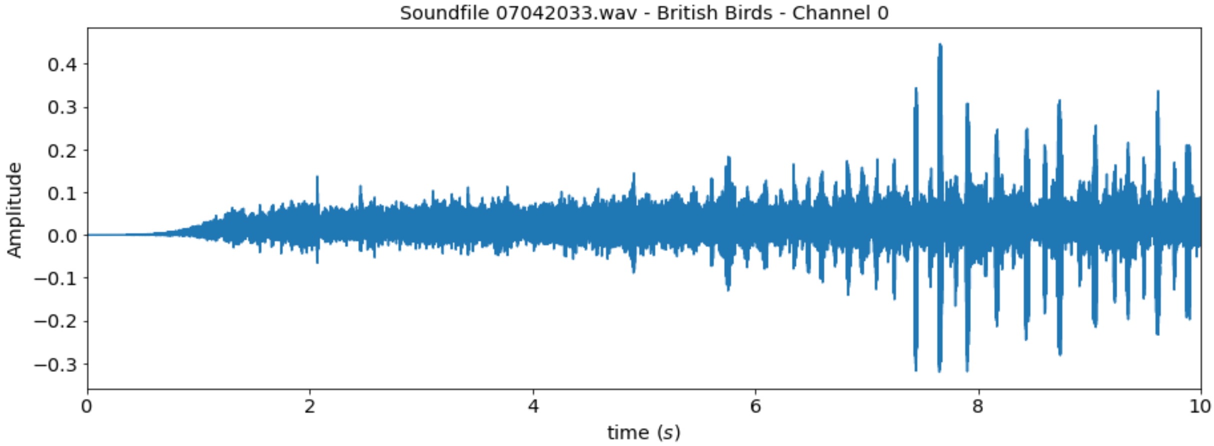
1. Using your function, compute spectral centroid features for contiguous 46 ms frames hopped 50% for the audio file you used in part 1. (Ignore any frames at the end of audio files that are less than that length.) Plot the first 10 seconds of your time domain waveform, and plot the series of spectral centroids you extracted.

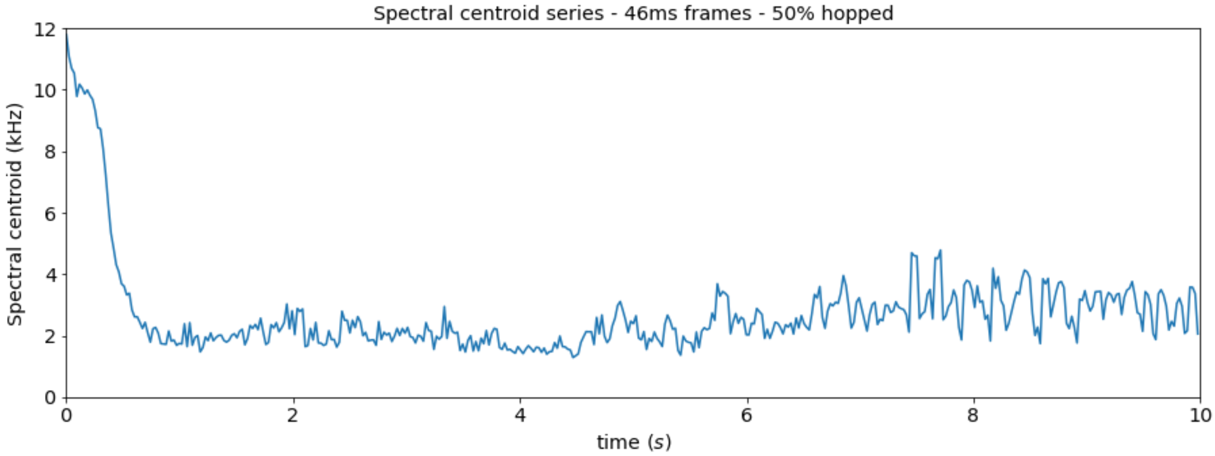
In [12]:

```
# add your code below

frame_size = 0.046 * sample_rate
hop_size = 0.5*frame_size

time = 10
time_size = int(time * sample_rate)

nwindows = int( (time_size - frame_size) / hop_size)

R = spectral_centroid(samples[ind], frame_size, hop_size, sample_rate)
R = R[:nwindows]

sample = samples[ind][:time_size]

time_axis1 = 1 / (sample_rate) * np.arange(0, time_size, 1)
time_axis2 = (time / nwindows) * np.arange(0, nwindows, 1)

# Plotting
fig1, ax1 = plt.subplots()
ax1.plot(time_axis1, sample)
ax1.set_xlim((0,10))
ax1.set_xlabel(r"time ($s$)")
ax1.set_ylabel(r"Amplitude")
ax1.set_title(f"Soundfile {filename} - British Birds - Channel {ind}")

fig2, ax2 = plt.subplots()
ax2.plot(time_axis2, R/1000)
ax2.set_xlim((0,10))
ax2.set_ylim((0,12))
ax2.set_xlabel(r"time ($s$)")
ax2.set_ylabel(r"Spectral centroid (kHz)")
ax2.set_title(f"Spectral centroid series - 46ms frames - 50% hopped")
```

Out[12]:

```
Text(0.5, 1.0, 'Spectral centroid series - 46ms frames - 50% hopped')
```

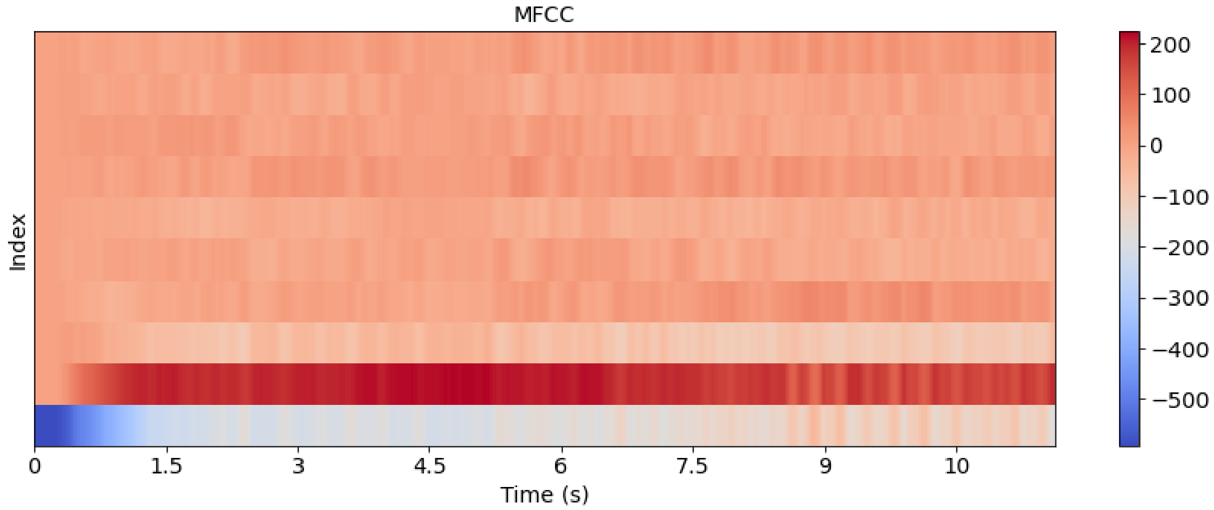




1. Using the librosa package (https://github.com/librosa), extract the first 10 MFCC features from your audio file using Hann windows of 25 ms duration and 10 ms hop size, and an FFT size of 8192 samples. Display the extracted MFCCs for the first 10 seconds.

In [13]:
```
# add your code below

hop_size1 = int(0.010 * sample_rate)
time = 10
time_size = int(time * sample_rate)

mfccs = librosa.feature.mfcc(y=samples[ind][:time_size],
                             sr=sample_rate,
                             n_mfcc=10,
                             n_fft=8192,
                             hop_length=hop_size1,
                             window='hann'
                             )


fig, ax = plt.subplots()
img = librosa.display.specshow(mfccs, x_axis='time', ax=ax, sr=sample_rate)
ax.set_ylabel("Index")
ax.set_xlabel("Time (s)")
fig.colorbar(img, ax=ax)
ax.set(title='MFCC')
```

Out[13]:
```
[Text(0.5, 1.0, 'MFCC')]
```


MFCC
Index
0
1.5
3
4.5
6
7.5
9
10
Time (s)
200
100
0
−100
−200
−300
−400
−500