Lab1 使用FIR滤波器分离鸟类声音

01: DSP & Python

> 该实验改编自Xilinx的DSP-PYNQ (https://github.com/Xilinx/DSP-PYNQ)，使用Xilinx Vitis HLS生成的FIR IP来进行音频滤波操作

检查我们的信号

`birds.wav` 是一个音频文件，包含两种鸟类的声音，其由Stuart Fisher录制，可以在此处 (https://www.xeno-canto.org/28039)获取。

In [38]:
```
from IPython.display import Audio
Audio("birds.wav")
```

Out[38]:

0:00 / 0:12

播放上面的音频，我们可以发现两种鸟类的叫声混杂在一起：

- 频率较低、鸣叫时长较短的鸟类是麻鹬
- 频率较高、鸣叫时长较长的鸟类是麻雀

我们希望通过数字信号处理的方法将两种鸟类的分离开来，并使用硬件函数为其加速。

Curlew
Photo by Vedant Raju Kasambe
Creative Commons Attribution-Share Alike 4.0
(https://creativecommons.org/licenses/by-sa/4.0/deed.en)

Chaffinch
Photo by Charles J Sharp
Creative Commons Attribution 3.0
(https://creativecommons.org/licenses/by/3.0/deed.en)

读入信号

让我们使用SciPy的 `wavfile` 模块完成信号的读入，fs为采样频率，aud_in为原始数据。

In [39]:
```
from scipy.io import wavfile

fs, aud_in = wavfile.read("./birds.wav")
```

我们查看下采样频率大小，原始数据的类型、长度和格式等信息。

采样频率为标准的44.1KHz，数据是以int16格式的numpy.ndarray保存的，共有554112个采样点。

In [40]:
```
print(fs)
print(type(aud_in))
print(len(aud_in))
print(aud_in.dtype)
print(aud_in[10000:10009])
```

```
44100
<class 'numpy.ndarray'>
554112
int16
[-37  25 128 210 183 202 278 310 300]
```

绘制频谱图

为了便于观察，我们可以创建一些交互式的绘图组件来帮助数据可视化。

在Notebook中绘制数据的频谱图是一个很好的选择，这有助于观察信号随时间的频率变化。

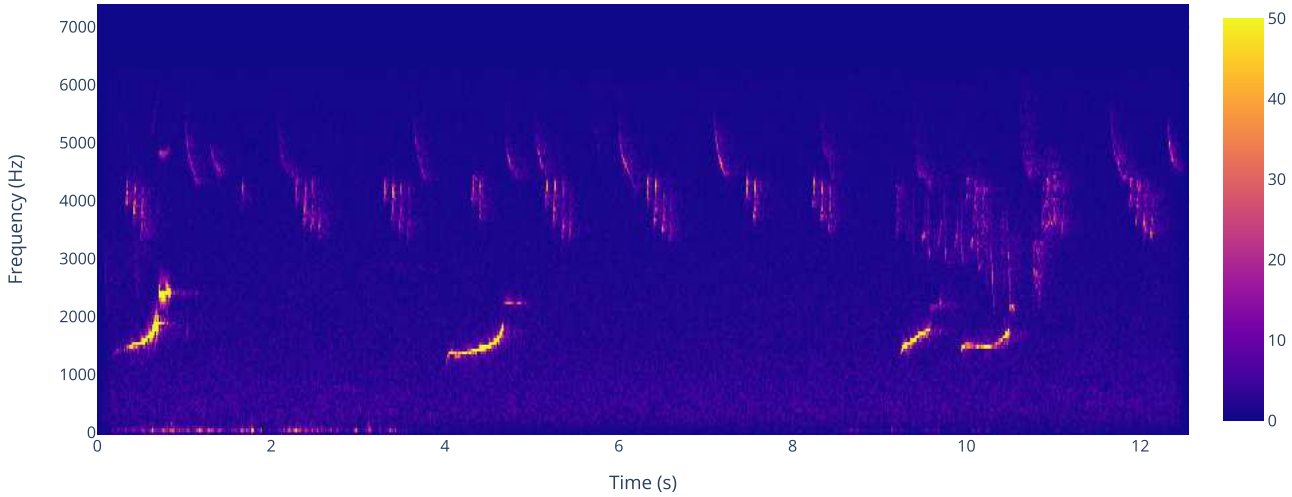
首先，我们禁止来自scipy的FutureWarnings，这些警告是针对Python包中将来将被弃用的特性的。

In [41]:
```
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

我们采用plotly的一些底层API来绘制图形，使用scipy的一些模块来获得频谱数据。

图形的绘制会需要一定的时间。

In [42]:
```
import plotly.graph_objs as go
import plotly.offline as py
from scipy.signal import spectrogram, decimate

def plot_spectrogram(samples, fs, decimation_factor=3, max_heat=50, mode='2D'):

    # Optionally decimate input
    if decimation_factor>1:
        samples_dec = decimate(samples, decimation_factor, zero_phase=True)
        fs_dec = int(fs / decimation_factor)
    else:
        samples_dec = samples
        fs_dec = fs

    # Calculate spectrogram (an array of FFTs from small windows of our signal)
    f_label, t_label, spec_data = spectrogram(
        samples_dec, fs=fs_dec, mode="magnitude"
    )

    # Make a plotly heatmap/surface graph
    layout = go.Layout(
        height=500,
        # 2D axis titles
        xaxis=dict(title='Time (s)'),
        yaxis=dict(title='Frequency (Hz)'),
        # 3D axis titles
        scene=dict(
            xaxis=dict(title='Time (s)'),
            yaxis=dict(title='Frequency (Hz)'),
            zaxis=dict(title='Amplitude')
        )
    )

    trace = go.Heatmap(
        z=np.clip(spec_data,0,max_heat),
        y=f_label,
        x=t_label
    ) if mode=='2D' else go.Surface(
        z=spec_data,
        y=f_label,
        x=t_label
    )

    py.iplot(dict(data=[trace], layout=layout))


plot_spectrogram(aud_in, fs, mode='2D')
```



观察上图，我们可以明显地区分开两种鸟类的声音——麻鹬声位于1.2-2.6kHz之间而麻雀声位于3-5kHz之间。

下面，我们可以先在Python中设计滤波器将**麻雀声**提取出来，即滤掉1.2-2.6kHz之间的麻鹬声。

FIR 滤波器

我们可以使用SciPy中的信号处理模块来设计FIR滤波器的参数与实现滤波功能：

- `firwin` 可用于计算满足设计要求的滤波器参数
- `freqz` 可用于计算滤波器的频率响应

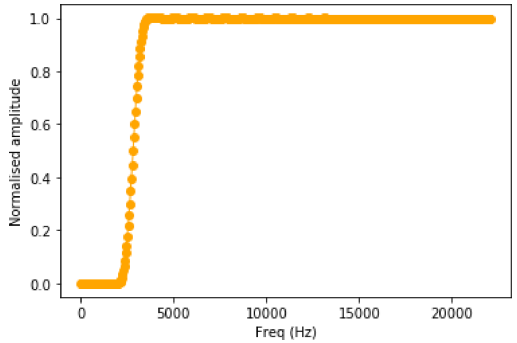
为了过滤出频率更高的麻雀声，需要一个高通滤波器来抑制2.6kHz以下的信号，我们预留些余量，将截止频率设置为2.8kHz。

In [43]:
```
from scipy.signal import freqz, firwin

nyq = fs / 2.0
taps = 99

# Design high-pass filter with cut-off at 2.8 kHz
hpf_coeffs = firwin(taps, 2800/nyq, pass_zero=False)

freqs, resp = freqz(hpf_coeffs, 1)

sample_freqs = np.linspace(0, nyq, len(np.abs(resp)))
```

我们可以对频率响应曲线进行可视化，帮助我们更好地进行滤波器设计。

In [44]:
```
import matplotlib.pyplot as plt

plt.plot(sample_freqs, abs(resp), linewidth=1, color="orange", marker="o")
plt.xlabel("Freq (Hz)")
plt.ylabel("Normalised amplitude")
```

Out[44]: Text(0, 0.5, 'Normalised amplitude')



滤波器共有99个参数，以float64的格式保存。

In [45]:
```
len(hpf_coeffs)
```

Out[45]: 99

查看部分参数。

In [46]:
```
hpf_coeffs[0:9]
```

Out[46]:
```
array([-3.33912973e-04, -1.58154938e-04,  5.64697472e-05,  2.92705880e-04,
        5.25410677e-04,  7.20013809e-04,  8.34195940e-04,  8.23730019e-04,
        6.52233264e-04])
```

02:DSP & HW

为了便于硬件实现，我们将原始的float64格式转化为int32，将其进行简单的量化。

In [47]:
```
hpf_coeffs_quant = np.array(hpf_coeffs)
hpf_coeffs_hw = np.int32(hpf_coeffs_quant/np.max(abs(hpf_coeffs_quant)) * 2**15 - 1)
```

最终将写入到IP中的系数如下。

In [48]:
```
hpf_coeffs_hw
```

Out[48]:
```
array([  -13,    -6,     1,     9,    18,    26,    30,    29,    23,
          10,    -8,   -32,   -55,   -75,   -86,   -82,   -62,   -25,
          26,    84,   140,   182,   200,   183,   129,    40,   -75,
        -202,  -317,  -398,  -424,  -377,  -254,   -58,   187,   452,
         693,   863,   915,   812,   532,    69,  -558, -1309, -2128,
       -2942, -3676, -4261, -4637, 32767, -4637, -4261, -3676, -2942,
       -2128, -1309,  -558,    69,   532,   812,   915,   863,   693,
         452,   187,   -58,  -254,  -377,  -424,  -398,  -317,  -202,
         -75,    40,   129,   183,   200,   182,   140,    84,    26,
         -25,   -62,   -82,   -86,   -75,   -55,   -32,    -8,    10,
          23,    29,    30,    26,    18,     9,     1,    -6,   -13])
```

加载Overlay

Overlay模块封装了ARM CPU与FPGA的PL部分进行交互的接口。

- 我们可以通过简单的 `Overlay()` 方法将刚才生成的硬件设计加载到PL上
- 通过 `overlay.fir_wrap_0` 语句，我们可以通过访问的Python对象的形式来与IP交互

In [49]:
```
from pynq import Overlay
overlay = Overlay("./fir.bit")
fir = overlay.fir_wrap_0
```

分配内存供IP使用

`pynq.allocate` 函数用于为PL中的IP分配可以使用的内存空间。

- 在PL中的IP访问DRAM之前，必须为其保留一些内存供IP使用，分配大小与地址
- 我们分别为输入、输出和权重三个部分分配内存，数据类型为int32
- `pynq.allocate` 会分配物理上的连续内存，并返回一个 `pynq.Buffer` 表示已经分配缓冲区的对象

In [50]:
```
from pynq import allocate
sample_len = len(aud_in)
input_buffer = allocate(shape=(sample_len,), dtype='i4')
output_buffer = allocate(shape=(sample_len,), dtype='i4')
coef_buffer = allocate(shape=(99,), dtype='i4')
```

将python的本地内存中的音频数据和系数数据，复制到我们刚分配的内存中。

In [51]:
```
np.copyto(input_buffer, np.int32(aud_in))
np.copyto(coef_buffer, hpf_coeffs_hw)
```

我们可以看到，缓冲区本质也是numpy数组，但是提供了一些物理地址属性。

In [52]:
```
input_buffer[10000:10009]
```

Out[52]:
```
PynqBuffer([-37,  25, 128, 210, 183, 202, 278, 310, 300])
```

In [53]:
```
coef_buffer[0:9]
```

Out[53]:
```
PynqBuffer([-13,  -6,   1,   9,  18,  26,  30,  29,  23])
```

In [54]:
```
coef_buffer.physical_address
```

Out[54]:
```
377790464
```

配置IP

我们可以直接使用IP的 `write` 方法，将刚分配的内存空间的地址写入到IP对应位置上

对于数据长度，我们可以直接在对应寄存器写入值。

In [55]:
```
fir.s_axi_control.write(0x1c,input_buffer.physical_address)
fir.s_axi_control.write(0x10,output_buffer.physical_address)
fir.s_axi_control.write(0x28,coef_buffer.physical_address)
fir.s_axi_CTRL.write(0x10, sample_len)
```

启动IP

控制信号位于0x00地址，我们可以对其进行写入与读取来控制IP启动、监听是否完成。

In [56]:
```
import time

fir.s_axi_CTRL.write(0x00, 0x01)
start_time = time.time()
while True:
    reg = fir.s_axi_CTRL.read(0x00)
    if reg != 1:
        break
end_time = time.time()

print("耗时: {}s".format(end_time - start_time))
```

```
耗时：0.021216392517089844s
```

结果已经被写入到了 `output_buffer` 中，我们可以进行查看

In [57]:
```
output_buffer[10000:10009]
```

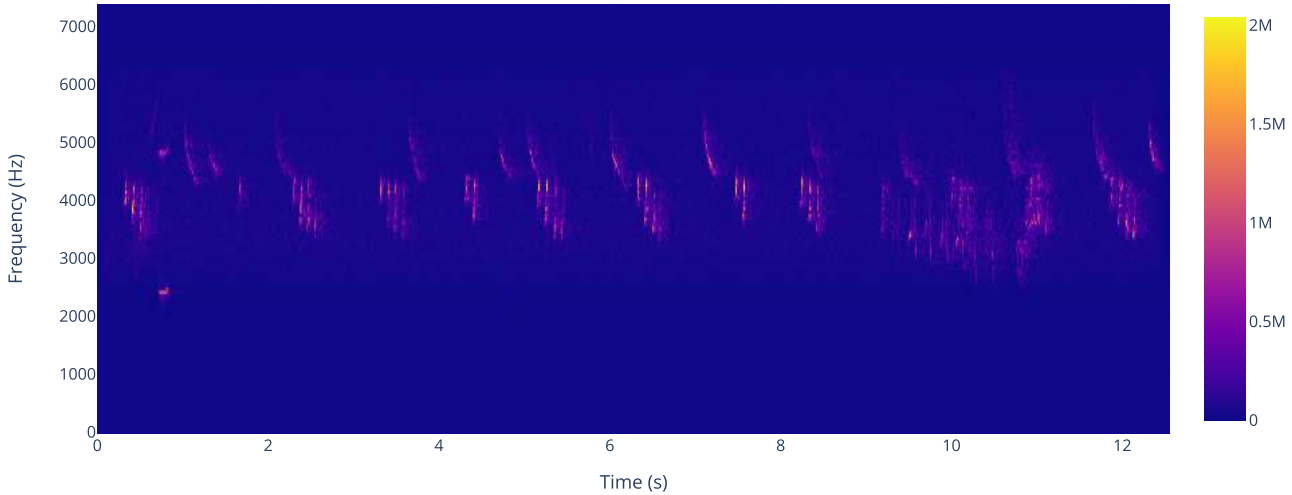
Out[57]:
```
PynqBuffer([-4340999,  2073422,  5686747,  -114713,  -862652,   491182,
             1283565,  4466583,  4497105])
```

可视化结果

仍然使用上述绘图组件，我们对硬件函数的结果进行可视化

- 可以看到，相较于原信号，低频部分都被较好的去除了
- 由于对参数进行了量化，值普遍偏大

In [58]:
```
plot_spectrogram(output_buffer, fs, mode='2D', max_heat=np.max(abs(output_buffer)))
```



我们可以再对输出结果进行缩放，将结果写入到音频 hpf_hw.wav 中并进行试听，可以发现麻鹬的声音已经被成功去除了。

In [59]:
```
from IPython.display import Audio

scaled = np.int16(output_buffer/np.max(abs(output_buffer)) * 2**15 - 1)
wavfile.write('hpf_hw.wav', fs, scaled)
Audio('hpf_hw.wav')
```

Out[59]:

0:00 / 0:12

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: