Digital Signal Processing Lab

Take Home Exam - Q2

Saad Zubairi
shz2020

November 8, 2025

Solution

Premise

Question 2 extends Question 1's solution to include vizualization of the frequency spectra of both original and changed audio in real time as the user slides the slider to change the alpha value.

Approach

For this question, I essentially used the code implementation from `prog_07_filter_spectrum.py`, Week 8's Demo 63 code. The main addition here is of matplotlib's animation framework to visualize the spectra. Of course, everything else from Q1 remains unchanged, but the processing loop structure was slightly modified to work with matplotlib's `FuncAnimation` instead of a simple while loop.

Detailed steps to solution

- **Matplotlib Visualization:** Following the structure from Demo 63, I set up a matplotlib figure with two subplots, one for each, input and output spectras. The figure is embedded in the Tkinter window using `FigureCanvasTkAgg`:

```
# DEFINE FIGURE
fig1 = matplotlib.figure.Figure()                    # not using pyplot
ax_inpFFT = fig1.add_subplot(2, 1, 1)
ax_outFFT = fig1.add_subplot(2, 1, 2)
fig1.set_size_inches((6, 8))  # (width, height)

# Initialize plot lines
X_in = np.fft.rfft(input_block)  # real fft
f1 = np.arange(X_in.size) / BLOCKLEN
[line_org_spectra] = ax_inpFFT.plot(f1, np.abs(X_in)/BLOCKLEN)
ax_inpFFT.set_xlim(0, 0.5)
ax_inpFFT.set_ylim(0, y_lim)
ax_inpFFT.set_title('Spectrum of input signal')
ax_inpFFT.set_xlabel('Frequency (cycles/sample)')

X_out = np.fft.rfft(output_block)  # real fft
f2 = np.arange(X_out.size) / BLOCKLEN
[line_con_spectra] = ax_outFFT.plot(f2, np.abs(X_out)/BLOCKLEN)
ax_outFFT.set_xlim(0, 0.5)
ax_outFFT.set_ylim(0, y_lim)
ax_outFFT.set_title('Spectrum of output signal')
ax_outFFT.set_xlabel('Frequency (cycles/sample)')

```

Snippet 1: Matplotlib figure setup

- **Processing Loop:** The main processing loop from Q1 was tweaked a bit to work with matplotlib's animation framework. Here are the changes made:

  1. The loop is now driven by `FuncAnimation` instead of a while loop
  2. The update function must return the plot line objects for the plotting
  3. The spectrum plots are updated each iteration by computing the FFT of both input and output blocks

```
def my_update(i):
    global frame_idx
    global prev_tail
    root.update()

    # get slider value in real time
    alpha_from_slider = alpha.get()

    # get next input frame
    input_block = frames[frame_idx]

    # process the scaled
    output_block = process_block_fft_scaling(input_block, alpha_from_slider)

    # overlap and add
    output_block[:len(prev_tail)] = 0.5 * (output_block[:len(prev_tail)] +
    prev_tail)

    # clipping
    output_chunk = np.clip(output_block[:Hop], -MAXVALUE, MAXVALUE)
    output_chunk = np.around(output_chunk).astype(np.int16)

    # write to the graph variables
    line_org_spectra.set_ydata(np.abs(np.fft.rfft(input_block))/BLOCKLEN)
    line_con_spectra.set_ydata(np.abs(np.fft.rfft(output_block))/BLOCKLEN)

    stream.write(output_chunk.tobytes())

    prev_tail = output_block[Hop:]
    # increment to the next frame (circular)
    frame_idx = (frame_idx + 1) % len(frames)

    return (line_org_spectra, line_con_spectra)

```

Snippet 2: Modified update function with visualization

The loop logic for calculation of course, as you can tell, remains exactly the same.

- **Animation:** The animation is initialized with `FuncAnimation`, which continuously calls `my_update()` at regular intervals (10 milliseconds for a smoother, faster updating animation but we can also increase the value for examining the spectra in close detail). This is basically what lets us have the smooth real-time updates for the spectrum visualization:

```
my_anima = animation.FuncAnimation(
    fig1,
    my_update,
    init_func = my_init,
    interval = 10,   # milliseconds
    blit = True,
    cache_frame_data = False,
    repeat = False
)

Tk.mainloop()

```

Snippet 3: Animation setup

- **Some other notable differences from Q1:**
  - The block size was also changed from 1024 to 256 samples, for a smoother animation.

Addendum

Here's the full implementation in code

```
import pyaudio
import wave
import numpy as np
import os
import math
import tkinter as Tk

# function to make a preprocessed list of frames for overlapped block processing
def frames_to_process_with_hops(all_frames, block_size, overlap_factor):
    frames = []
    hop_count = math.floor(block_size * (1 - overlap_factor))
    idx_i = 0
    idx_l = block_size
    while idx_l < len(all_frames) - 1:
        frames.append(all_frames[idx_i:idx_l])
        idx_i += hop_count
        idx_l = idx_i + block_size
    return frames

# functiuon for processing blocks with scaling via fft and ifft with
    interpolation
def process_block_fft_scaling(input_block, alpha):
    X = np.fft.rfft(input_block)
    Y = np.zeros_like(X)
    for src_ind in range(X.size):
        dst_ind = src_ind / alpha
        if dst_ind < X.size - 1:
            i0 = int(np.floor(dst_ind))
            i1 = i0 + 1
            t = dst_ind - i0
            Y[src_ind] = (1 - t) * X[i0] + t * X[i1]
    y = np.fft.irfft(Y)
    return y



base_dir = os.path.dirname(os.path.abspath(__file__))
wavfile = os.path.join(base_dir, 'author.wav')
wf = wave.open(wavfile, 'rb')
#output_wavfile = 'author_output_blocks_corrected.wav'

#print('Play the wave file %s.' % wavfile)

# Open wave file (should be mono channel)
#wf = wave.open( wavfile, 'rb' )

CONTINUE = True # Variable for the looping mechanic
CHANNELS        =  wf.getnchannels()
RATE            = wf.getframerate()
WIDTH           = wf.getsampwidth()
signal_length   = wf.getnframes()
BLOCKLEN        = 1024
OVERLAP_FACTOR  = 0.5
MAXVALUE        = 2**15 - 1

```

```
print('The file has %d channel(s).'              % CHANNELS)
print('The frame rate is %d frames/second.'     % RATE)
print('The file has %d frames.'                  % signal_length)
print('There are %d bytes per sample.'          % WIDTH)

#output_wf = wave.open(output_wavfile, 'w')      # wave file
#output_wf.setframerate(RATE)
#output_wf.setsampwidth(WIDTH)
#output_wf.setnchannels(CHANNELS)

Hop = int(BLOCKLEN * (1 - OVERLAP_FACTOR))
binary_data = wf.readframes(signal_length)
all_samples = np.frombuffer(binary_data, dtype=np.int16)

root = Tk.Tk()
root.title('Real-time Frequency Scaling')

# Scaling factor init
alpha = Tk.DoubleVar()
alpha.set(1.0)
# print(alpha.get())

# Slider config here
alpha_slider = Tk.Scale(root, label='Scaling Factor (From 0.5 to 1)', variable=
    alpha, from_=0.5, to=2.0,resolution=0.01, orient=Tk.HORIZONTAL, length=300)
alpha_slider.pack(side=Tk.TOP)

# Quit button config here
def handle_close_quit():
    global CONTINUE
    CONTINUE = False
B_quit = Tk.Button(root, text='Quit', command=handle_close_quit)
B_quit.pack(side=Tk.BOTTOM, fill=Tk.X)

# Pyaudio config
p = pyaudio.PyAudio()
# Open audio stream
stream = p.open(
    format      = p.get_format_from_width(WIDTH),
    channels    = CHANNELS,
    rate        = RATE,
    input       = False,
    output      = True )


# Main loop
print('* Start')

prev_tail = np.zeros(BLOCKLEN - Hop)
frames = frames_to_process_with_hops(all_samples, BLOCKLEN, OVERLAP_FACTOR)
frame_idx = 0
while CONTINUE:
    root.update()

    # get slider value in real time
    alpha_from_slider = alpha.get()

    # get next input frame
    input_block = frames[frame_idx]
```

```
113
114     # process the scaled
115     output_block = process_block_fft_scaling(input_block, alpha_from_slider)
116
117     # overlap and add
118     output_block[:len(prev_tail)] = 0.5 * (output_block[:len(prev_tail)] +
        prev_tail)
119
120     # clipping
121     output_chunk = np.clip(output_block[:Hop], -MAXVALUE, MAXVALUE)
122     output_chunk = np.around(output_chunk).astype(np.int16)
123     stream.write(output_chunk.tobytes())
124
125     prev_tail = output_block[Hop:]
126     # increment to the next frame (circular)
127     frame_idx = (frame_idx + 1) % len(frames)
128
129 print('* Finished')
130
131
132 stream.stop_stream()
133 stream.close()
134 p.terminate()
135 wf.close()
```