Sava Jankovic 400292525
Parth Kotecha 400303078

FouierTransform.py

For the take home exercise regarding the fourier transform, we had to create a period square waves of arbitrary duty cycles, which would be projected at execution. They would be plotted in both time and frequency domain. We had to revisit our knowledge on duty cycles, and implemented various functions to implement it into our code.

For starters, the duty that is shown on the graph should be generated as a value between 0 and 1, which was implemented through random.uniform(0,1). Henceforth, every time the code was run, a random value would be generated. Additionally, to graph the cycle, the interval was set, along with the dt variable which is 1 over the sampling rate which in our case was 100. The dutycyclesquare variable was assigned to the square signals being generated by the duty cycle. Finally, we would first plot the time spectrum and then finish it off by plotting a frequency spectrum with the sampling rate and the dutycyclesquare function, with the DFT function building on it.

filterDesign.py

This part of the lab required more complex filtering which involved three different signals that we have used from the very first part. The three signals (tone 1, tone 2 and tone 3) all had different frequencies (20,30 and 40 respectfully). Since the task required us to filter out two of the three, we created an algorithm with manual cut offs, being 25 and 35.

Since the cutoffs are 25 and 35, this would result in the first and the third tones being filtered out. The variable was also applied to the equation cutoffeq, which used the equation provided in the lab report to generate a filter for the different tones. Finally, when the sigcomb collated all the tones in one, we used the firwin function from scipy to create a graphic with the filtered results. As a final touch, the freqzPlot provided the final graph, with a text which provides the cutoff frequencies which can be adjusted by the user.

blockProcessing.py

The first function, convolution, is an implementation of a convolution operation between a set of coefficients (coeff) and a set of data (data), with a specified size and buffer. The function returns two outputs: output_buffer, which contains a subset of the input data, and filtered_data, which contains the result of the convolution operation. The blockFilter function applies a low-pass filter to a given audio data (audio_data) by processing it in blocks of a specified size (block_size). The function uses the audio_Fc and audio_Fs parameters to calculate the filter coefficients (coeff) using the lowpassfilter function and the number of taps N_taps. The filtered audio data is stored in filtered_data array.

The function loops through the audio data, processing it in blocks of the specified size and applying the convolution operation on each block using the convolution function. The buffer from the previous block is passed as an input to the next block, and the output buffer is updated with the new buffer. The result is a filtered version of the audio data with the low-frequency components remaining and the high-frequency components removed.