3DY4 Lab 1 Report

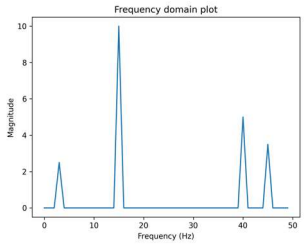
Jan 2024

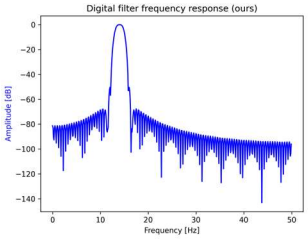
Kyler Witvoet, 400393313

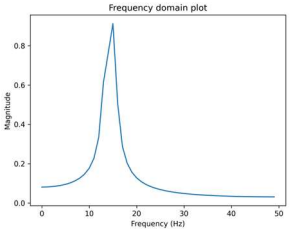
Josh Umansky, 400234265

TH1: Expanding on the implementation of the fourierTransform.py, we were tasked to develop a function similar to **'generateSin',** but for square waves. As suggested we completed this by using the **'Scipy.signal.Square'** method. The resulting function works in almost exactly the same as **'generateSin'**, but the phase parameter was substituted for dutyCycle, which has a default value of 0.5 and controls the duty cycle of the square wave.

TH2: To ensure our understanding of how a bandpass filter operated, we were asked to create a bandpass filter that would only allow the second lowest of a four-tone signal would pass through the filter. To accomplish this, we first had to create a signal that would abide by the requirements, which we accomplished by using the **'generateSin'** function that was created in the fourierTransform.py file. This four-tone signal was created with varying frequencies (3 Hz, 15 Hz, 40 Hz, 45 Hz). We then created the filter coefficients using the **'Scipy.signal.firwin'** function, with the cutoff frequencies of (Fc1 = 13 and Fc2 = 15), coupled with the sampling frequency of (Fs = 100Hz) these cutoff frequencies were established to ensure that the filter would only allow a signal with a frequency of ~15 Hz to pass and filter out any signals with lower or higher frequencies. Our filter signal (fx) was created by running **'Scipy.signal.lfilter'** to run the coefficients against our sin signal (x). We then plotted both signals in time and frequency domains to ensure the signals were filtered properly.







Before *Filter* *After*

TH3: We were tasked with developing our own implementations for the '**firwin'** and **'lfilter'** functions from the SciPy library, to be used in block processing of the audio signal. We used our custom "lowpass" function previously developed during the lab experiment to compute the filter coefficients, effectively replacing the functionality of the '**firwin'** function. For '**lfilter'** we used the convolution property of the Fourier transform. Initially we used our custom functions for the Fourier transform and its inverse to mimic the functionality of convolution [idft(dft(x)dft(h)) = x * h]. Once we confirmed that this approach worked, we switched to using the fast Fourier transform from NumPy to increase the computation speed. To use these functions for block processing we created a new function "our_block_processing", most of the code in this function is the same as "block_processing" but with our functions substituted in place of '**firwin'** and '**lfilter'**. The biggest challenge we faced was ensuring the continuity of the audio signals across blocks. To address this, instead of incorporating "filter_state" as lfilter does, we implemented a 50% overlap between consecutive blocks. We saved only the non-overlapping portion of each block, effectively emulating the role of 'filter_state' in smoothing the audio output. This strategy proved effective in maintaining audio continuity, with the half-overlap technique providing a balance between computational efficiency and signal integrity.