EE3025 ASSIGNMENT- 1

Ritwik Sahani - EE18BTECH11038

Download all python codes from

https://github.com/ritvix23/EE3025−IDP−DSP/tree/main/Assignment1/codes

And Latex-tikz codes from -

https://github.com/ritvix23/EE3025−IDP−DSP/tree/main/Assignment1

1 Problem

The command

```
output_signal = signal.lfilter(b,a,
    output_signal)
```

in Problem 2.3 is executed through following difference equation

$$\sum_{m=0}^{M} a(m)\, y(n-m) = \sum_{k=0}^{N} b(k)\, x(n-k) \tag{1.0.1}$$

where input signal is $x(n)$ and output signal is $y(n)$ with intial values all 0. Replace **signal.filtfilt** with your own routine and verify

2 Solution

This can be done using two different algorithms. Let us discuss both of them one by one.

2.1 Algorithm 1

A simple way could be to implement literally what equation 1.0.1 does, that is, find the output sequence sample by sample using previously calculated samples. However, the in-built routine **signal.filtfilt** does more than this. Upon checking the official documentation, it turns out that the in-built routine implements a forward-backward filter. So, let's first understand what a forward-backward filter is and why we need it.

2.1.1 Forward-Backward Filter: When we apply a filter, say a third order lowpass Butterworth filter, to an input signal, the filter introduces a phase lag at the output. This was the forward application of the filter, in a sense that we applied the filter to the input sequence just the way it was. In order to nullify this phase lag, we apply the same filter again at the output sequence obtained after forward pass, but after inverting the output sequence in time domain. This is the backward pass. The output from the backward pass is inverted again to give the final output of the forward-backward filter. The following equations should make it clear mathematically -

$$v(n) = (h * x)(n) \tag{2.1.1}$$

here, $v(n)$ is the ouptput of the first filtering operation, namely the forward pass, $x(n)$ is the input sequence, $h(n)$ is the filter impulse response. For the backward pass we invert/flip $v(n)$, so say v = [1, 2, 3], $FLIP(v)$ = [3, 2, 1]. Now,

$$w(n) = (h * FLIP(v))(n) \tag{2.1.2}$$

$w(n)$ is the output sequence from the second pass. It is inverted again to get final output -

$$y(n) = (FLIP(w))(n) \tag{2.1.3}$$

As we discussed earlier, one straight-forward algorithm to implement this filtering procedure is to use loops and calculate each output sample one by one. Rearranging equation 1.0.1 as -

$$a(0)\, y(n) = \sum_{k=0}^{N} b(k)\, x(n-k) - \sum_{m=1}^{M} a(m)\, y(n-m) \tag{2.1.4}$$

$$y(n) = \frac{\sum_{k=0}^{N} b(k)\, x(n-k) - \sum_{m=1}^{M} a(m)\, y(n-m)}{a(0)} \tag{2.1.5}$$

Arrays can be used to store past N+1 samples of input signal and past M samples of output signal, and these can be updated iteratively.

The soundfile generated from this algorithm can be found at -

https://github.com/ritvix23/EE3025−IDP−DSP/tree/main/Assignment1/soundfiles/Sound_from_algo1.wav

2.2 Algorithm 2

Another way could be to convert the sequences to z-domain, apply corresponding operations, and convert back.
Let' continue from equation 2.1.3 as -

$$y(n) = (FLIP(w))(n) \tag{2.2.1}$$

$$y(n) = FLIP(h * FLIP(h * x))(n) \tag{2.2.2}$$

Take z-transform both sides -

$$Y(Z) = H(z^{-1})H(z)X(z) \tag{2.2.3}$$

Where we used,

$$\mathcal{Z}\{h\} = H(z) \tag{2.2.4}$$

$$\implies \mathcal{Z}\{FLIP(h)\} = H(z^{-1}) \tag{2.2.5}$$

$H(z)$ can be obtained by taking z-transform both sides in equation 1.0.1,

$$Y(z)\sum_{m=0}^{M} a(m)z^{-m} = X(z)\sum_{k=0}^{N} b(k)z^{-k} \tag{2.2.6}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} b(k)z^{-k}}{\sum_{m=0}^{M} a(m)z^{-m}} \tag{2.2.7}$$

Where we used,

$$\mathcal{Z}\{x(n-k)\} = z^{-k}X(z) \tag{2.2.8}$$

$$\mathcal{Z}\{y(n-m)\} = z^{-m}Y(z) \tag{2.2.9}$$

The above algorithm can be implemented using built-in **fft** command to convert sequences to z-domain, and **ifft** command to convert them back to time domain.
The soundfile generated from this algorithm can be found at-

https://github.com/ritvix23/EE3025−IDP−DSP/tree/main/Assignment1/soundfiles/Sound_from_algo2.wav

The soundfile generated from builtin routine can be found at-

https://github.com/ritvix23/EE3025−IDP−DSP/tree/main/Assignment1/soundfiles/Sound_from_builtin_routine.wav

Single code file containing both the above algorithms can be found at -

https://github.com/ritvix23/EE3025−IDP−DSP/tree/main/Assignment1/codes/ee18btech11038.py

3 Comparison

Algorithm 1 takes $O(L(N+M))$ time, where L is the length of the input sequence. This is because L samples are calculated, and calculating each sample requires $N+M$ multiplications.
Algorithm 2 takes $O(Llog(L))$ time, which is same as time taken by **fft** algorithm.

4 Verification

We will use norm between the time sequences as a metric to calculate how 'different' they are.
The norm between the output sequence from original **signal.filtfilt** and output sequence from algorithm 1 turns out to be **127.025**.
While that between **signal.filtfilt** and algorithm 2 turns out to be **0.053**(which is quite small!).
Possible explaination for larger value for algorithm 2 is that the original routine **signal.filtfilt** does numerous other normalizations under the hood, which result in small deviations in the output samples from our algorithm.
Further, time domain and frequency domain response are plotted for perceptual verification.

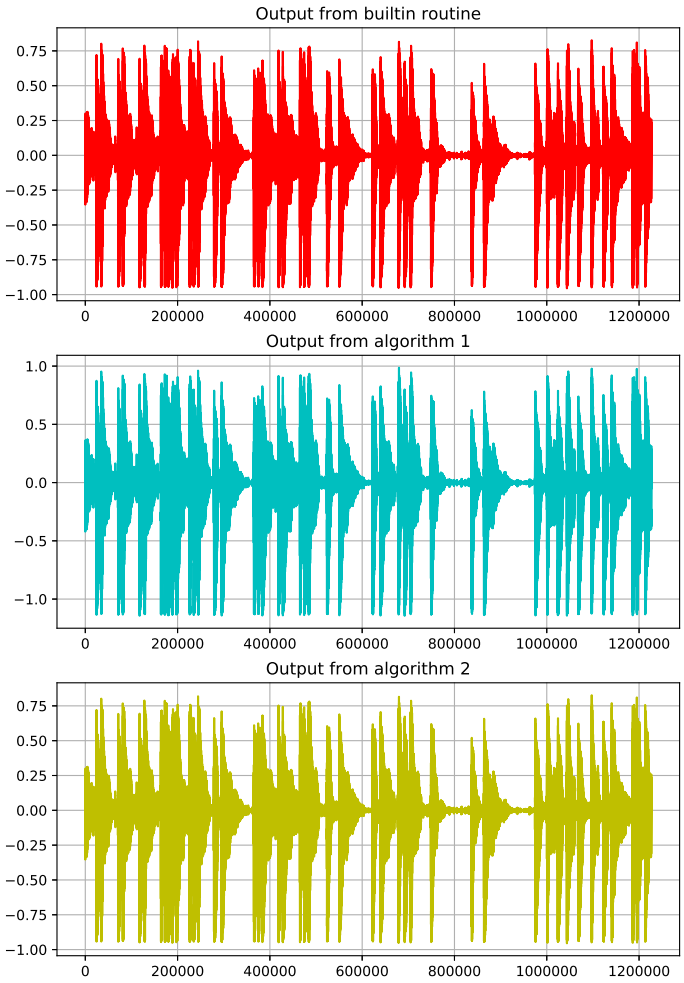

Fig. 0: Time domain response

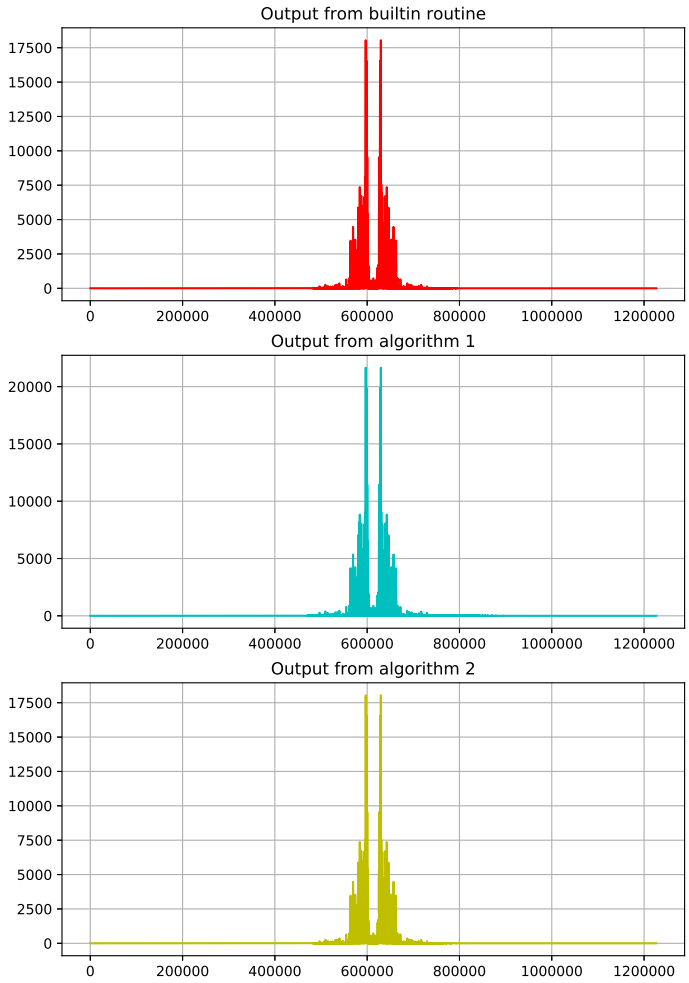

Fig. 0: Frequency domain response