**Part 1**

*Method*

To create a process that will approximate a sine wave with minimal SFDR at an arbitrary, I first consider the phase values which are input to the sinusoidal function over the first 50 periods of the signal. This value is expressed by , where is the fractional frequency. If we let so that the phase value begins at 0, we can iterate over the rest of the phases and find the point at which the phase best approximates 0 again. This will tell us the point at which the signal can be ‘re-read’ with very little distortion because the transition from ‘0’ back to the beginning of the digital readout (technically, back to the second sample, after zero) will be the most seamless. I make the assumption that a minimized distortion of phase over will result in a minimal SFDR distortion.

*Time Domain Results*

Figure 1 shows the result using this method to find the best index at which to repeat for the case of fractional frequency

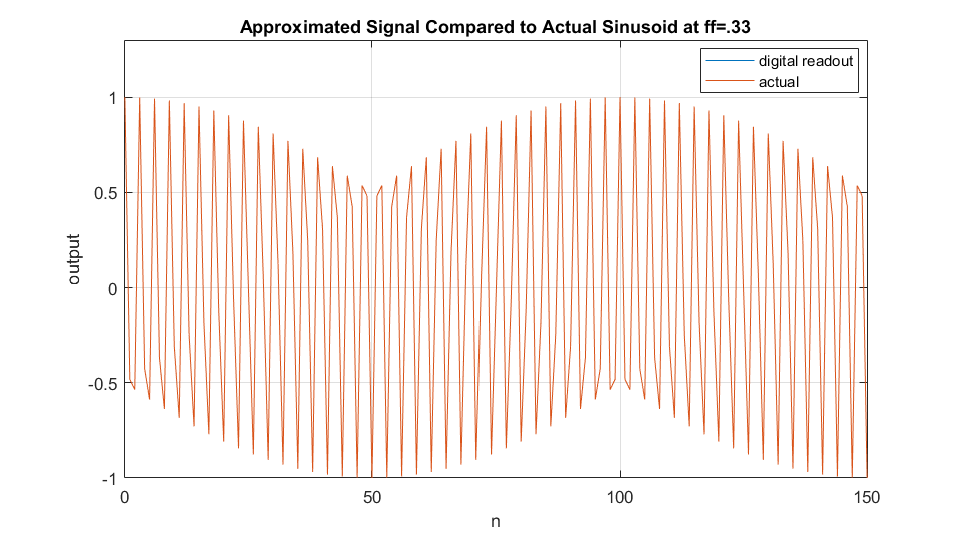


Figure 1 - Time-domain readout compared to the sinusoid it approximates, ff=.33

From Figure 1, we see that the algorithm has successfully selected a repetition point which succeeds in imitating a true sinusoidal signal at least for a short duration of 150 samples. Figure 2 shows the error of the signal compared to a true sinusoid for the first 500 periods of that sinusoid.

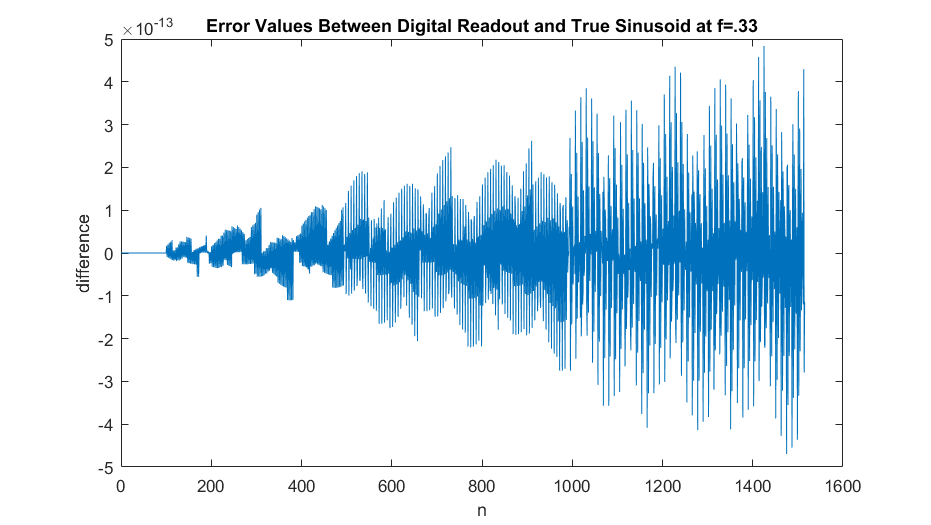


Figure 2 - Digital readout error

Figure 2 shows that the error of the digital readout system is quite small in magnitude, although it accumulates over time. This makes sense considering that the phase of the generated signal gradually drifts from that of the target signal every time the readout is repeated.

*Frequency Domain Results*

Figure 3 presents the resulting signal in the time domain using both Hamming and Blackman windows.

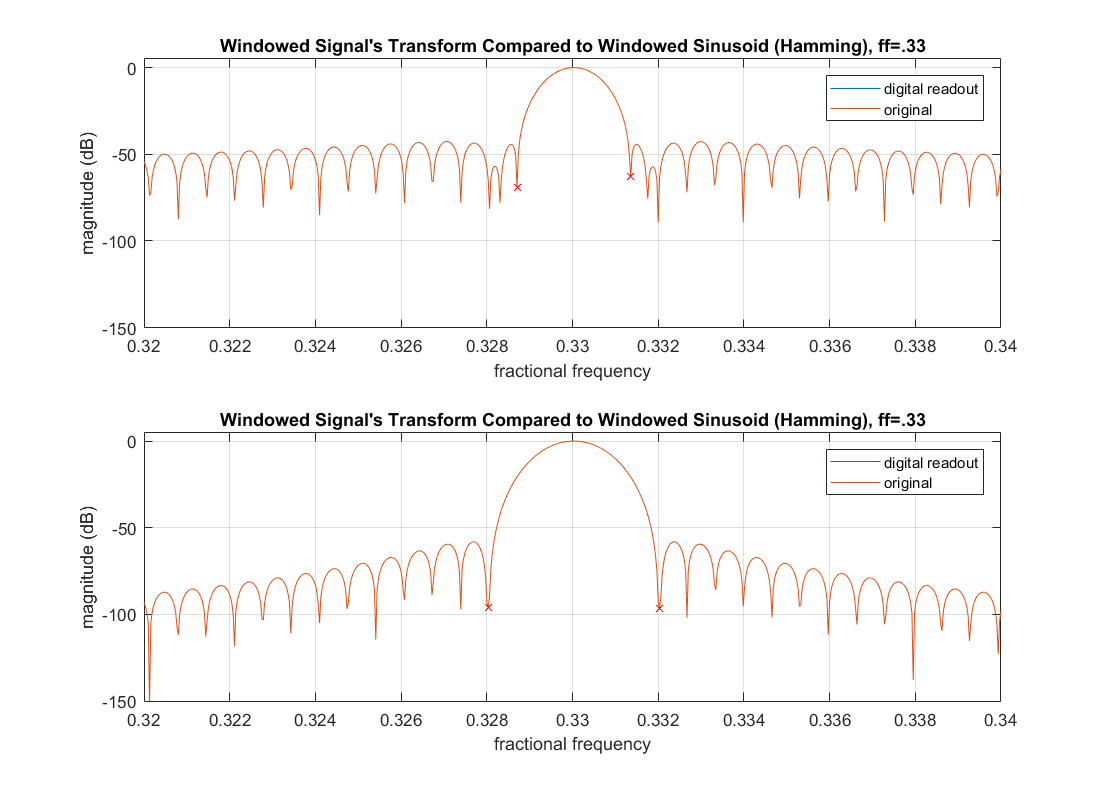


Figure 3 - Comparing the frequency domains using different windows

First, we can make some observations about the window shapes themselves. We know that the plots shown in Figure 3 are just scaled and shifted versions of the windows represented in the frequency-domain. This is because we multiplied the windows by a sinusoidal signal and then took the DFT of the result. This operation corresponds to convolving the transform of the window against the transform of a sinusoid which is a scaled and shifted pair of delta functions. The result of this convolution is a pair of scaled and shifted windows. Figure 3 focuses on the frequencies around , so the other transform of the window at is not visible. We can see that the Hamming window has a very narrow main lobe compared to the Blackman window, while the Blackman window drops off much faster and farther than the Hamming window does once outside of the main lobe.

*Determining SFDR*

It is worth noting that, for , the readout signal so closely approximates the original that the effects of the windows are more prominent than the signal’s distortion. If we were to find the SFDR of this signal naively by comparing the main lobe against the sidelobes, we would be essentially measuring a property that is inherent to the windowing function and not the signal of interest.

Instead, we can first calculate the distortion power that is a result of the digital readout approximation by taking the magnitude of the differences between the window frequency representation and the signal frequency representation.

The two red ‘X’ marks on each subplot of Figure 3 denote the boundaries of the main lobe as determined by locate\_mainlobelims, a custom function I wrote (see accompanying files). This utility will assist in determining the SFDR by enabling me to make a distinction between the main lobe and the sidelobes.

My method for determining the SFDR is simply to take the maximum value within the main lobe and compare it against the maximum value among all frequencies of the distortion signal which fall outside of the main lobe range. The distortion signal is calculated by finding the difference between a true windowed sinusoid and the windowed digital readout signal. Figure 4 shows the resulting distortion signal, determined using both the Hamming and Blackman window types.

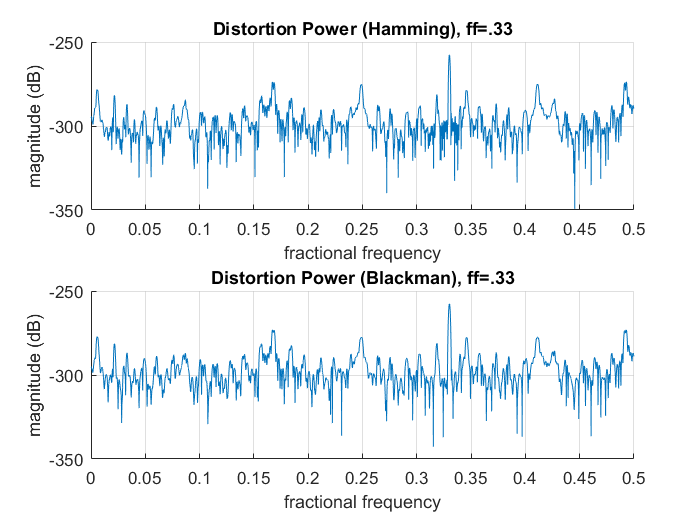


Figure 4 - The frequency content of the distortion signal

Interestingly, the fractional frequency of the sinusoid is frequency at which the distortion power is highest. Although this is true, we should avoid using the main lobe fractional frequency range in order to measure SFDR.

The output of my code provides the calculated SFDR of the generated signal for both the Hamming and Blackman windows.

SFDR\_HAMMING =

273.7088

SFDR\_BLACKMAN =

273.0951

The consistent results of this calculation appear to show that this method of calculating SFDR is impacted very little by the windowing method used. This would suggest that the calculated SFDR of roughly 273 dB is inherent to the signal.

**Part 2**

To calculate the total distortion of the signal, we have only to find the power of the distortion signal shown in Figure 4. Because these signals vary slightly depending on the window used, I will calculate the total distortion for both windows. To find the power of the signal represented by the frequency domain, as in Figure 4, we can use Parseval’s theorem which shows how to find the power of the signal when it is presented in the frequency domain:

To take advantage of this result, I will iterate through each frequency bin, taking the square of the magnitudes at the bin. At then end, I should have the power as long as I divide by the length of the entire array. Also, in my case, since I am only working with the positive frequencies, I must multiply my resulting power by a factor of two to get the actual power of the error signal. After dividing the power of the error signal by the power of the targeted sinusoid, I will take this power to be my total distortion. Using this method, I get the following result:

TD\_BLACKMAN =

-282.6392

TD\_HAMMING =

-283.4232

**Part 3**

Figure 5 shows both the total distortion and the spurious free dynamic range for a set of 100 random fractional frequencies. In addition, the SFDR and TD for is included. Note the difference between the two y axes.

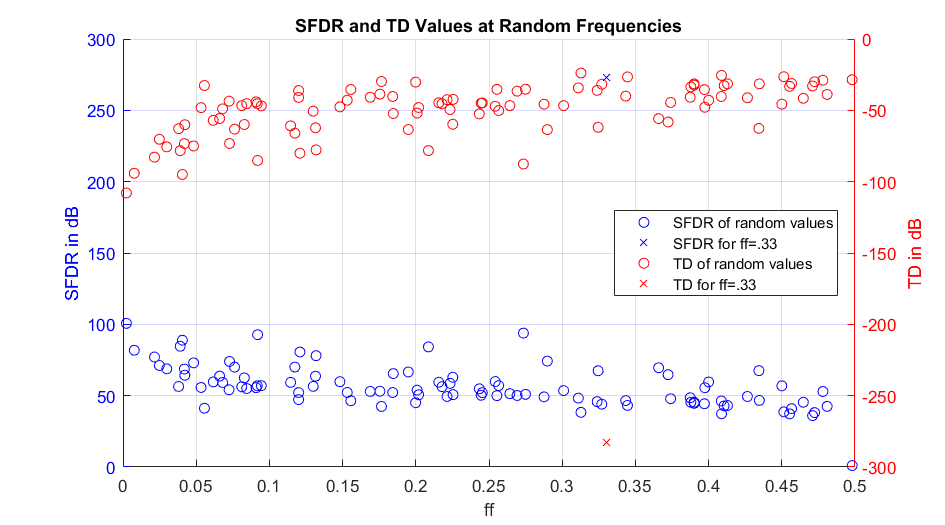


Figure 5 - SFDR and TD of random values plotted over fractional frequency

The most noticeable fact illustrated in Figure 5 is that the point is a hugely outstanding outlier with extremely high SFDR and extremely low TD that is unlike any of the randomly generated points. I assume that this is because .33 is a ‘more’ rational number than the other numbers that were randomly generated. As an example, the fractional frequency with the lowest SFDR is . A sinusoid with a fractional frequency of .33 has a phase that can be expressed as or .

If we let be 100, we obtain a phase that is divided evenly by , satisfying the requirements stated earlier. By contrast, there is no such small value of that will satisfy the same conditions for . This is why I believe all of the signals with randomly generated frequencies perform far worse than a signal with a ‘nice’ fractional frequency.

Another interesting trend shown in Figure 5 is that the TD notably increases and SFDR notably decreases as fractional frequency increases. I believe this is because we were permitted to search within the space of 50 periods to find a solution that best minimizes distortion. Naturally, signals of a lower fractional frequency have more samples per period. As a result, my algorithm had more points over over which to search for a suitable point to repeat the sinusoid.

Finally, Figure 5 shows that the SFDR and the TD seem to ‘mirror’ one another in a peculiar way. Qualitatively, this makes some since because I would expect the TD and SFDR to be inversely related to one another. I am surprised at how exact the mirroring seems to be, however.

**Part 4**

Now with a LUT that is limited to 1024 memory locations, the search space for the best point to repeat the sinusoid is no longer a function of period length. This will likely hamper the ‘advantage’ that the lower frequency signals had in Part 3. Figure 6 shows the results.

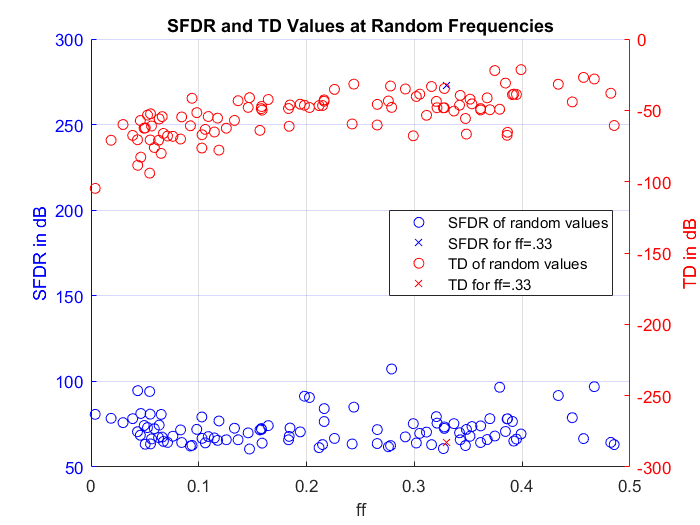


Figure 6 - SFDRs when the lookup table is limited to 1024 samples

As expected, the SFDR has changed from trending down over ff to having little slope at all. Forcing a constant-length lookup table means that signals of higher frequency are able to iterate for more than 50 periods which increases the chances that we will find a more optimal point at which to repeat the sinusoidal signal. Oddly, the mirroring of Figure 5 is far less evident in Figure 6. This may be in part due to the fact that my algorithm reduces the amount of zero-padding that is done when the original signal is reduced in size. This may mean that some resolution is lost in the frequency domain representation, in turn leading to less precise mirroring of the values.

**Part 5**

*Direct Convolution*

Figure 7 illustrates the operations that take place during the steady-state convolution process. The grey arrows represent elements of the input being multiplied by elements of the UPR and being accumulated in the output vector. The circled ‘y’ value represents an output value that has been fully calculated.

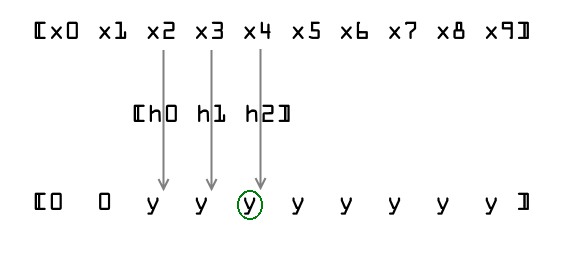


Figure 7 - Illustration of direct convolution with MAC operations illustrated in grey and the final output in green

In Figure 7 we can see that direct convolution requires 3 MACs per output sample for this example. From this figure, we can further guess that the number of MACs per output sample are exactly the length of the UPR during steady-state operation. The UPR signal provided with the project has a length of 255 so we will work with the assumption that direct convolution requires 255 MAC operations per output sample.

The following sections describe the behavior of the overlap-save and overlap-add algorithms during steady state as described in lecture and as illustrated in Leland B. Jackson’s DSP book (2nd edition). The two algorithms take two input signal streams and process them simultaneously, producing two outputs.

*Overlap Save*

1. Take one signal block from each stream, each of length N. We will refer to these blocks as and . This step requires no MAC operations.
2. Run the FFT algorithm for the complex signal . Because the FFT is a linear operation, we can expect the output to be where and are the results of taking the FFT of the respective time-domain signals on their own. This operation requires MAC operations.
3. Multiply the zero-padded, N-length transform of the UPR with the ‘combined’ transform . We will call the resulting vector . Because this operation is complex, it will probably equate to more than N MACs. I managed to find literature on a MAC implementation that claims to perform complex MAC operations with twice the duration as a regular MAC, so I will assume that this operation takes 2N MACs.
4. Run the IFFT algorithm for the complex vector *.* Because the IFFT is a linear operation, we can expect the output to be where and are the results of taking the IFFT of the respective frequency-domain vectors on their own. This operation requires MAC operations.
5. To extract the vectors and , we need to take the real and imaginary parts of the vector . If we assume that the complex values are stored in rectangular form as an ordered pair, this operation should require no MAC operations, as the real and imaginary parts can be read directly from the memory space.
6. For both vectors, discard the first M samples, where M is the length of the UPR. This is because the FFT-based circular convolution process makes these first values invalid. This operation should not require any MAC operations.
7. The remaining samples are valid output values which can be sent as the next samples in the output stream. In total, we have 2(N-M) output samples.

Figure 8 illustrates a more concise and visual description of the overlap-save method

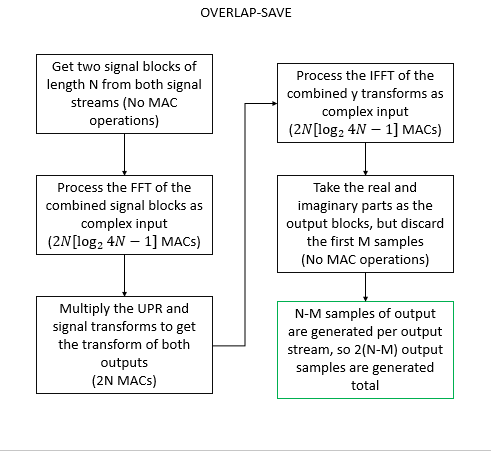


Figure 8 – Summary of overlap-save algorithm with simultaneous input processing

Adding up all the MAC operations for and dividing by the total number of output samples during this process, we have

*Overlap Add*

Figure 9 illustrates the overlap-add algorithm.

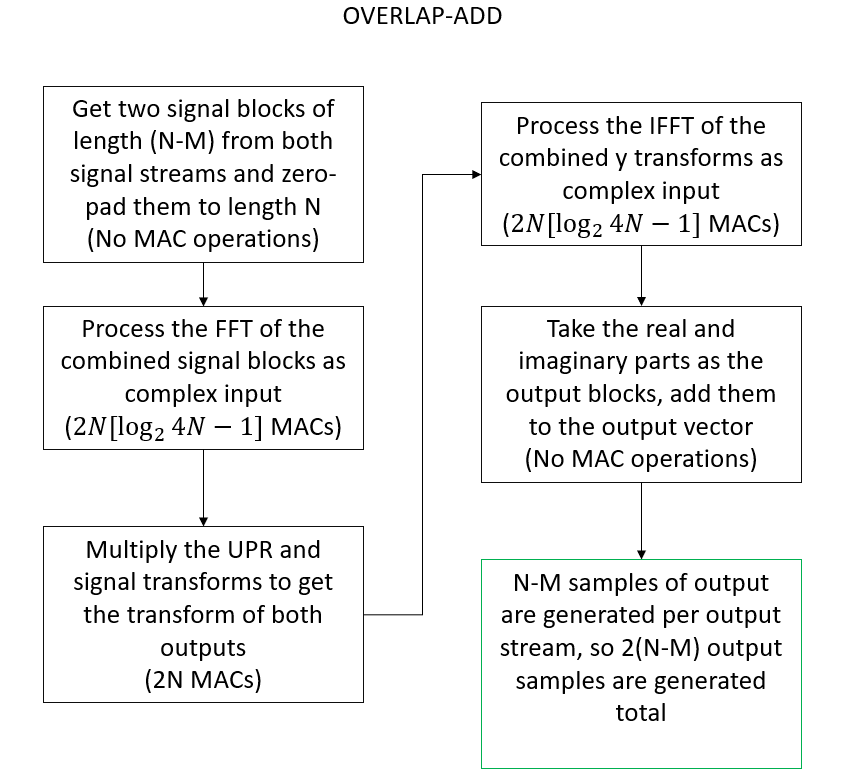


Figure 9 - Summary of overlap-save algorithm with simultaneous input processing

As shown, the algorithm only differs from the way that the input stream is taken as input and the way that output values are put into the output stream. There is no difference in the number of MACs per output sample overall.

*Performance as a function of window size*

Figure 10 shows the theoretical performance of both the direct convolution method and the overlap-add, overlap-save methods.

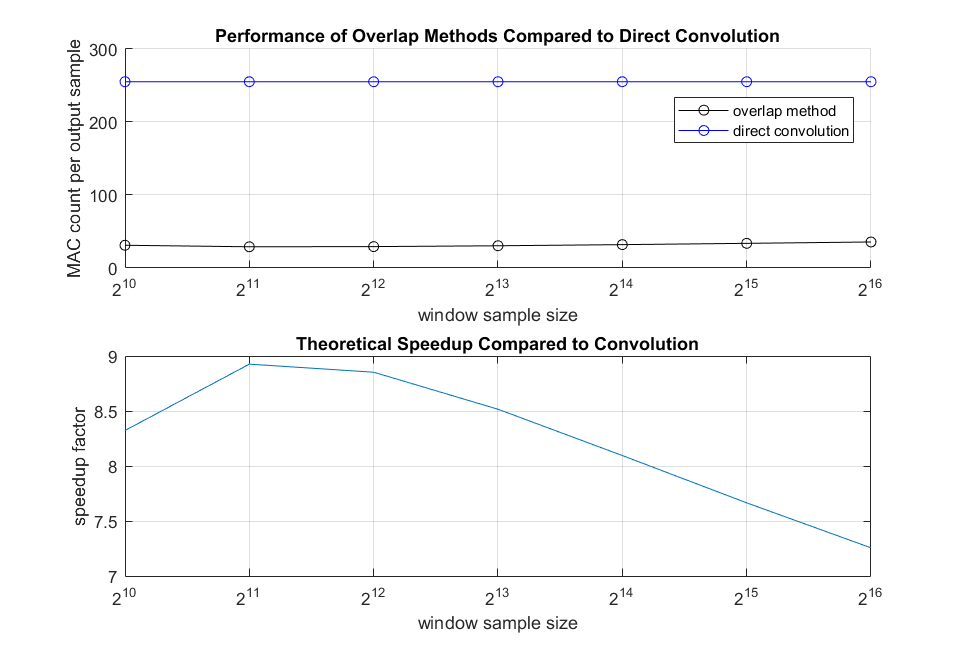


Figure 10 - Calculated performance of different filtering algorithms for varying window sizes

Of course, since the direct convolution MAC count is solely a function of the UPR length, it is constant as the window sample size varies. The overlap-add and overlap-save methods do vary slightly over window size, however with an optimum window size at 211 with 28.56 MACs per output sample and a speedup factor of 8.93.

*Performance as a function of UPR length*

Figure 11 shows the speedup factor as calculated previously as a function of both the window size and the UPR length. The speedup factor is taken in dB in order to make the changes over UPR length more visible for the smaller window sizes. The differences are difficult to distinguish otherwise.

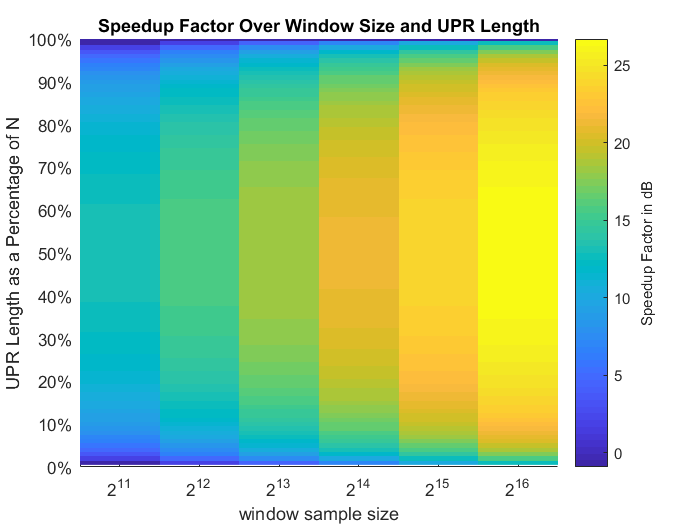


Figure 11 - Speedup factor (taken in dB) over various UPR lengths and window sizes

Figure 11 makes it quite clear that the optimal UPR length for all window sizes we see here is half of that same window size. It is interesting to note that as the window size increases, the speedup factor becomes far higher than it does for smaller window sizes. This means it was not the increased window size that lead to a decreased speedup factor in Figure 10. This decrease can instead be attributed to the ever smaller UPR length in comparison to the increasing window size.

**Part 6**

To show that the filtering process described above functions as intended, we will analyze the results shown below for both the overlap-save and overlap-add methods. For these first sections, we use an .

*Overlap-Save*

See files p6.m, overlap\_save.m, and complex\_fft.m for the code used to generate these results.

In order to prove that the algorithm I wrote can process two signals simultaneously, I split up the given signal, xn, into two signals of equal length. Figure 12 shows the result of both direct convolution and my overlap-save algorithm on the first half of the signal. Figure 13 shows the same results for the second signal.

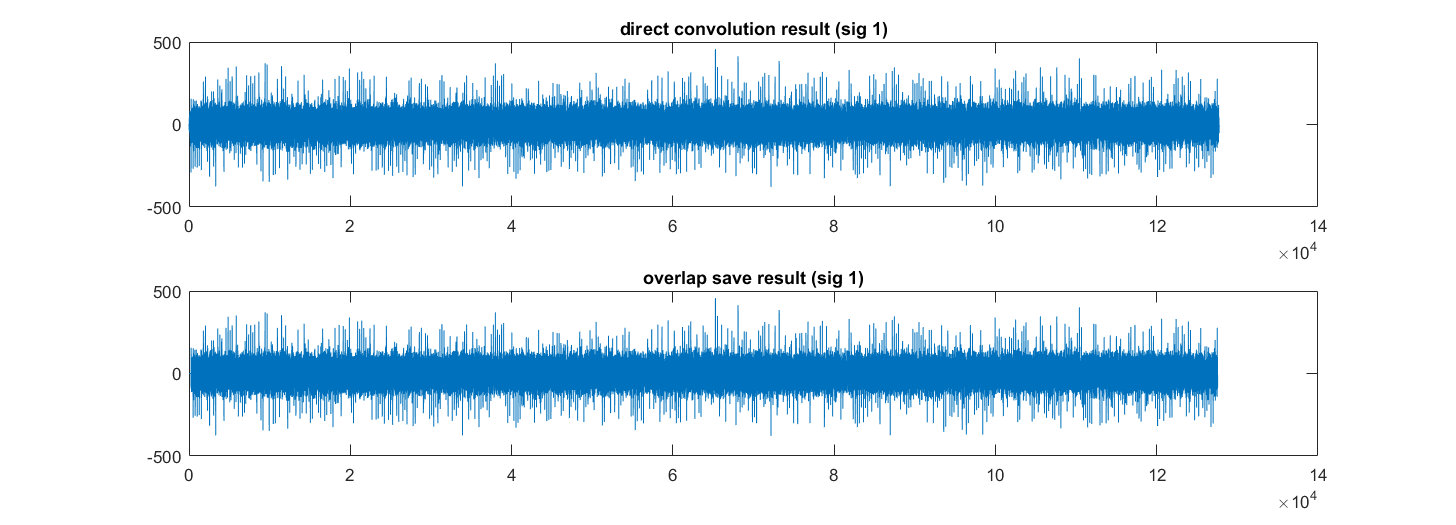


Figure 12 - Comparing output-save results with direct convolution results (signal 1)

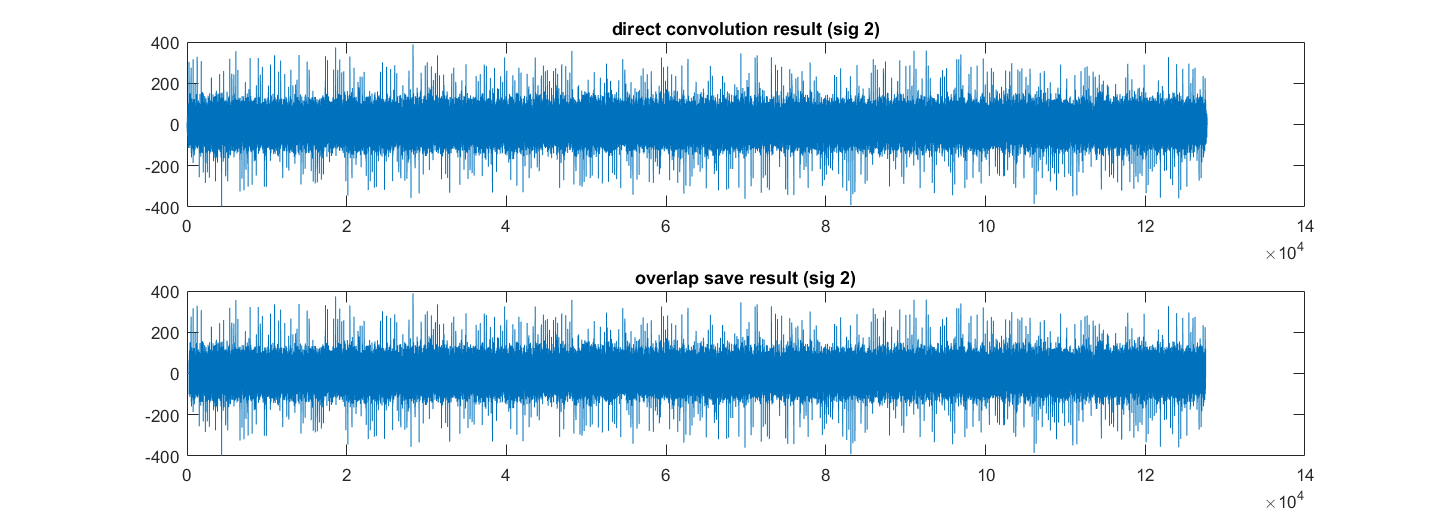


Figure 13 - Comparing output-save results with direct convolution results (signal 2)

The two Figures show that the overlap-save algorithm appears to do a very good job of imitating the behavior of direct convolution. Upon closer inspection, however, we can see some inconsistencies between the two signals at the start and end of the outputs. Figure 14 focuses on the differences at the beginning by showing only the first 2000 samples.

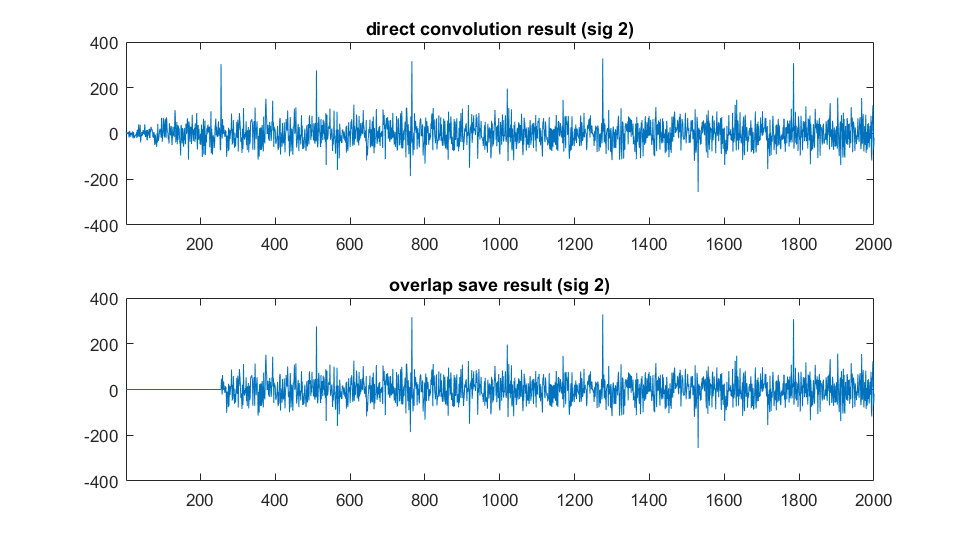


Figure 14 - Differences between the two outputs on 'power-up'

Figure 14 shows that the first 255 samples of the signal have been discarded. This is exactly the behavior that we expect the overlap-save algorithm to have because the first 255 samples are unusable.

Figure 15 focuses on the differences at the end by showing only the last 1000 samples.

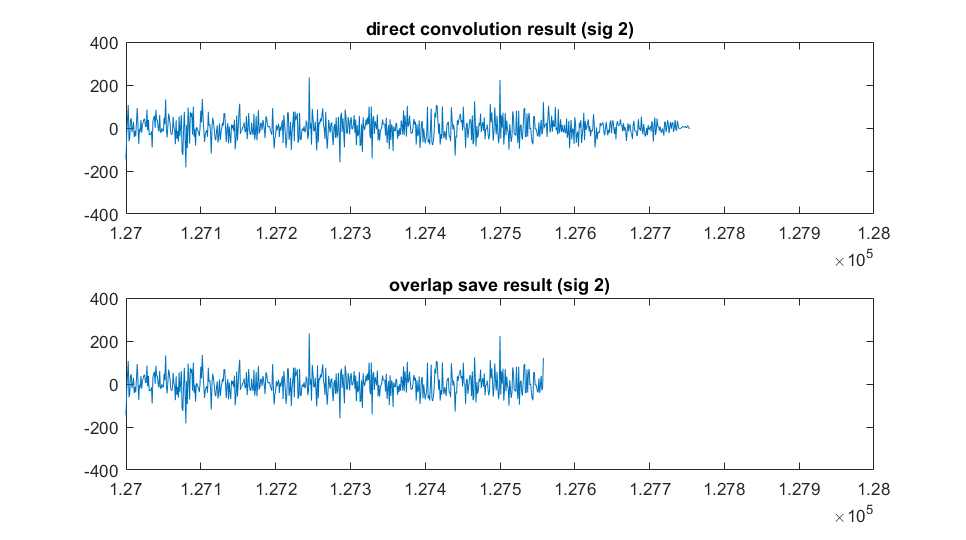


Figure 15 - Differences between the two outputs on 'power-down'

We can see from Figure 15 and from the MATLAB output below that the lengths of the two signals differ in length from each other and from the original input signal.

>> length(x1)

ans =

127500

>> length(y2)

ans =

127558

>> length(dir\_conv\_1)

ans =

127754

Because the length of a convolution is known to be N+M-1, the length of the convolution signal is no surprise since the length of the UPR is 255 and 127500+255-1 is exactly 127754. The fact that the overlap-add result is also slightly larger than the input signal is still no surprise because the algorithm will continuously process blocks of the input until it encounters the end of the signal. At this point it will zero-pad the remaining fraction of a block into its full size and process the result. This process is bound to result in an output that is slightly larger than the original input, and this is what we observe.

With these differences in mind, we can measure the success with which the overlap-save algorithm imitates the direct convolution process by ignoring the start and end portions of the output. The result is as follows:

>> max(y1(256:127558)-dir\_conv\_1(256:127558))

ans =

4.5475e-13

Given that the output signal could sometimes reach values as high as 450, an error on the order of 10-13 is acceptable and can be attributed to rounding.

*Overlap-Add*

See files p6.m, overlap\_add.m, and complex\_fft.m for the code used to generate these results.

In order to prove that the algorithm I wrote can process two signals simultaneously, I split up the given signal, xn, into two signals of equal length. Figure 16 shows the result of both direct convolution and my overlap-save algorithm on the first half of the signal. Figure 16 shows the same results for the second signal.

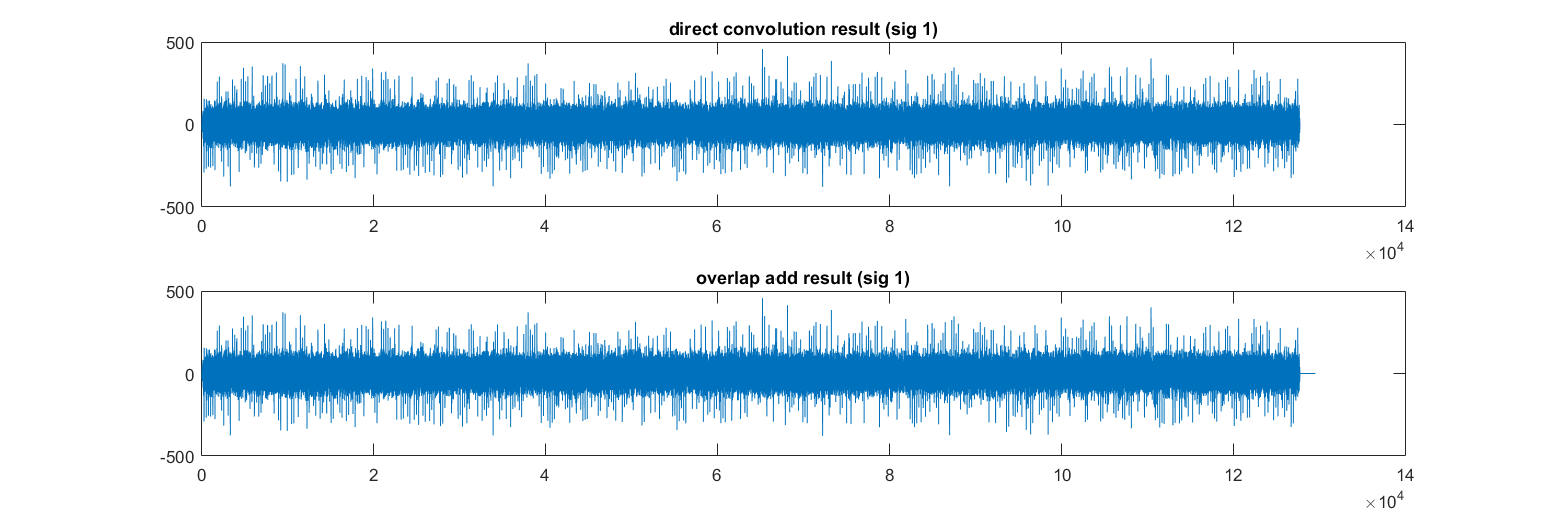


Figure 16 - Comparing output-add results with direct convolution results (signal 1)

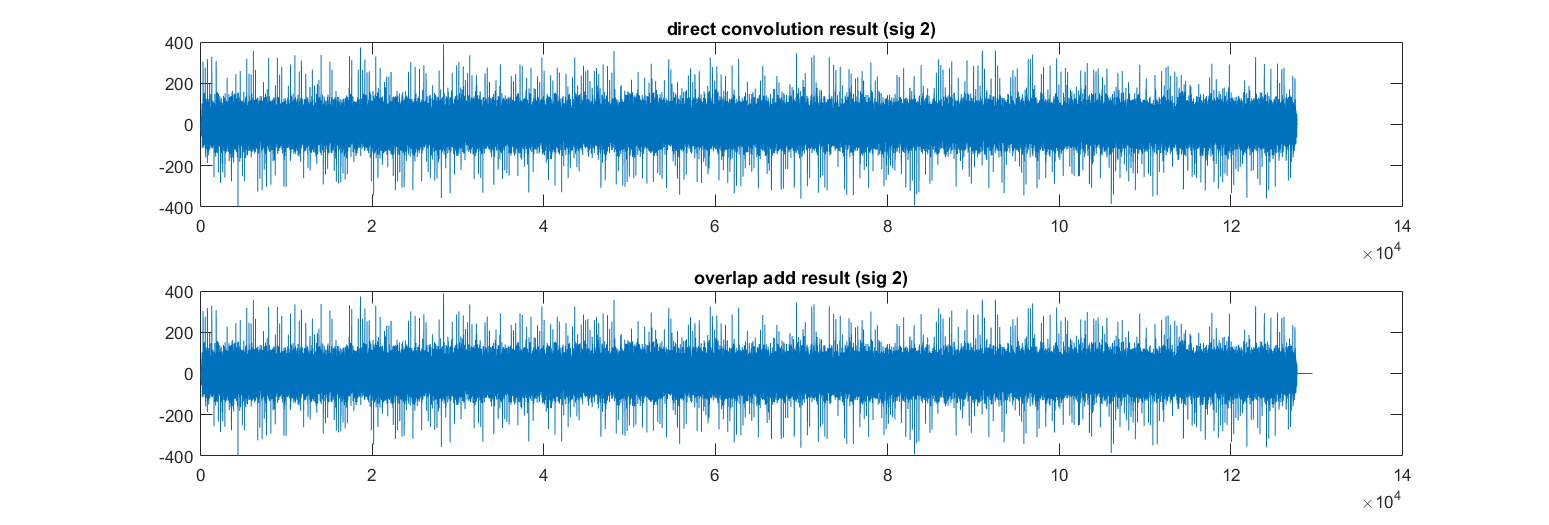


Figure 17 - Comparing output-add results with direct convolution results (signal 2)

The two Figures show that the overlap-save algorithm appears to do a very good job of imitating the behavior of direct convolution. Figure 18 focuses on the two filter results for signal 2 at the beginning by showing only the first 2000 samples.

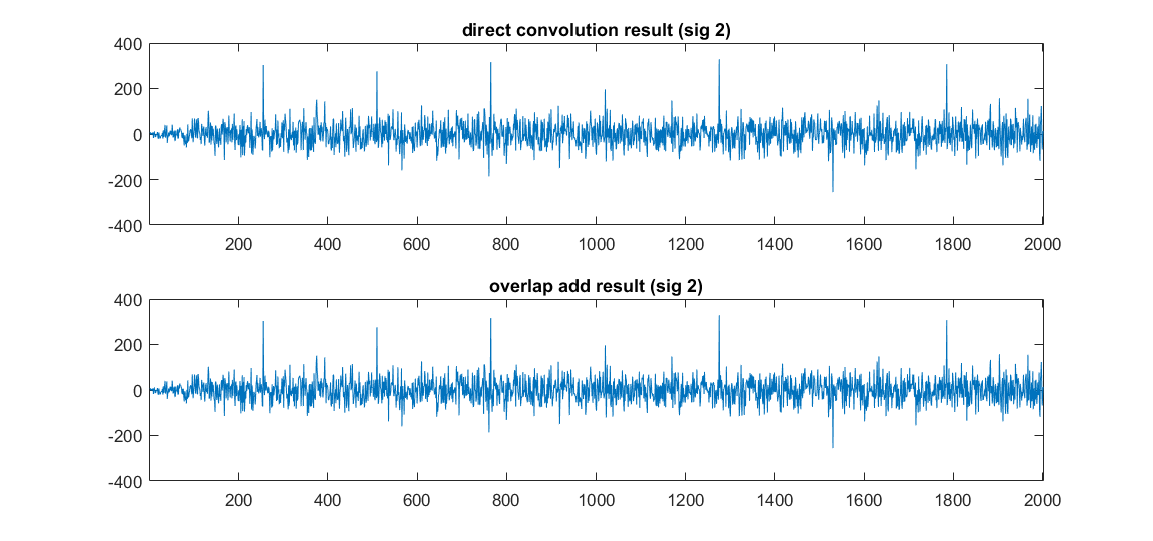


Figure 18 - Differences between the two outputs on 'power-up'

In contrast with the overlap-save method, the overlap-add result appears to entirely coincide with the direct convolution result from the very beginning. This makes some sense, as the overlap-add method does not require that the first values be discarded, unlike the overlap-save method.

Figure 18 focuses on the differences at the end by showing only the last 1000 samples.

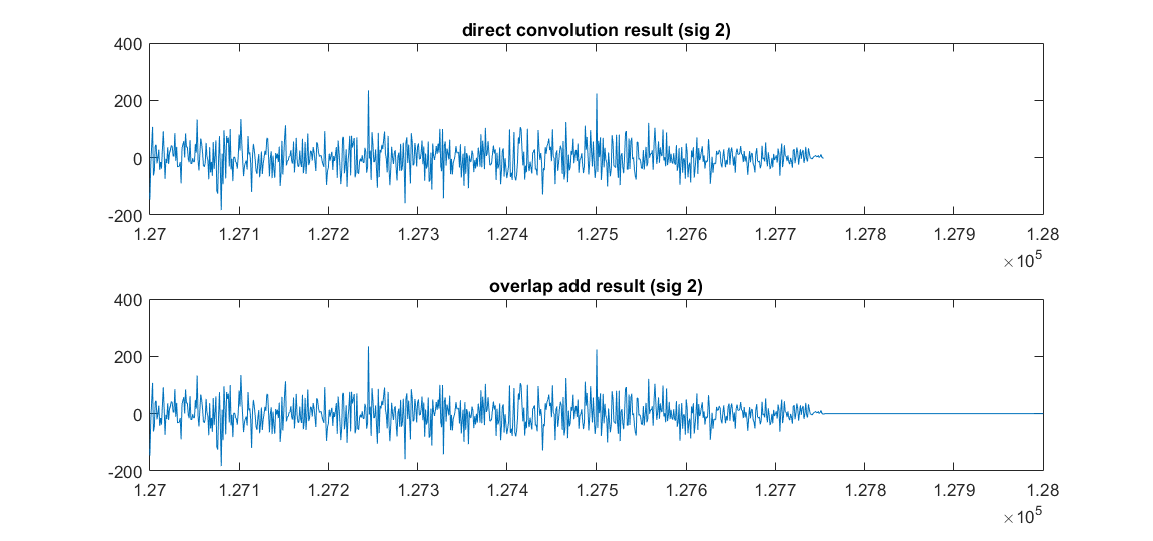


Figure 19 - Differences between the two outputs on 'power-down'

From Figure 19, we see that the two filter outputs are almost identical. The fact that the overlap-add result includes trailing zeros is only a symptom of the final block of data having trailing zeros when it was processed. If we disregard these trailing zero values, we can consider these signals to be of identical length. This is result is in contrast with that of the overlap-save method. we can measure the success with which the overlap-add algorithm imitates the direct convolution process by ignoring the trailing zeros. The result is as follows:

>> length(dir\_conv\_1)

ans =

127754

>> max(y1(1:127754)-dir\_conv\_1(1:127754))

ans =

4.5475e-13

Interestingly, this error value is exactly the same as the one found in the overlap-save section. In a sense, this makes sense because the core FFT process by which the output values are found is the same even if the way in which they are read and written to and from the x and y vectors differ.

*Measuring the duration of the filtering algorithms*

Before making measurements of the filtering algorithms, I made sure to disable threading on MATLAB because I do not assume anything about multithreaded behavior during the previous calculation section. Figure 20 shows the results of timing the overlap-save, overlap-add, and direct convolution functions when processing the given input vector with window sizes varying from 211 to 216.

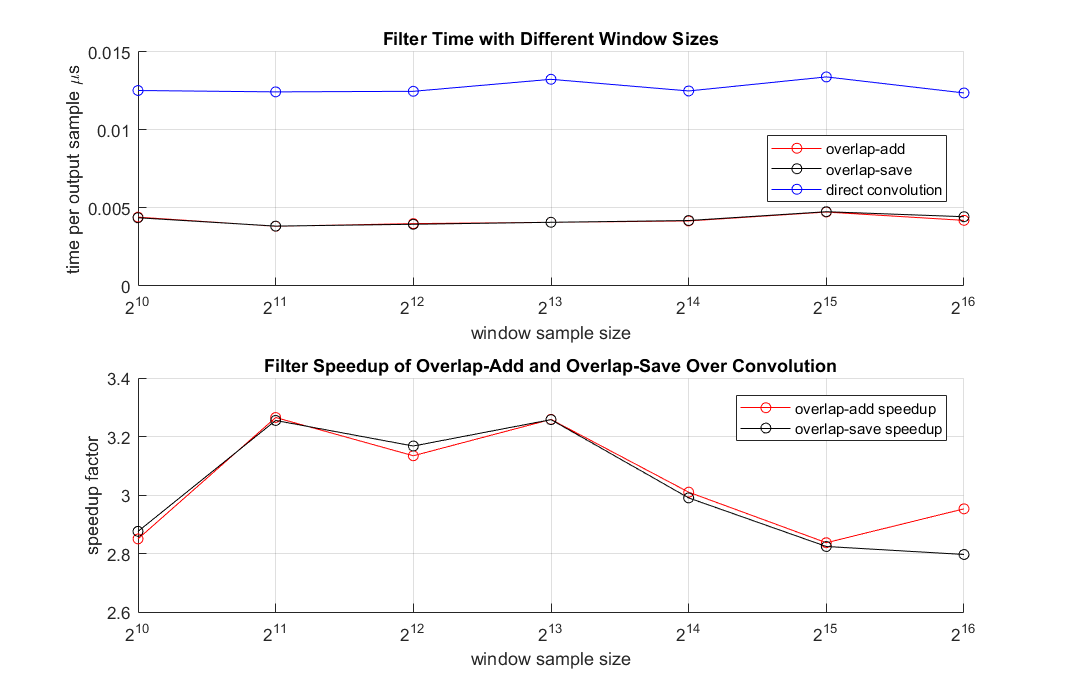


Figure 20 - performance of various filtering algorithms over a range of window sizes

The results do somewhat contradict my findings in Part 5. We can see that even the duration of the convolution varies somewhat when the window size is varied even though the convolution process does not change with the windowing size. As a result, I am not confident in the reliability of this data. Regardless, the speedup amount is not what I expected. We can see that the optimal window size seems to be 211 with a speedup factor of 3.05 instead of the theoretically calculated 8.27. Although the difference in optimal window size compared to the one calculated can be explained by the chaotic changes in process duration, probably due to the background process running on my laptop, the consistent difference in speedup factor cannot be so easily dismissed. It is possible that I missed a major consideration of the overlap-save and overlap-add algorithms which impacted the accuracy of my theoretical values. It is more likely that there is some internal MATLAB optimization of the conv function which makes the speedup factor of the overlap-add and save algorithms less than expected by comparison.

**Part 7**

At first glance, there is nothing about the input signal that stands out at the points where the output value reaches a large maximum value. Looking at the time-domain impulse response, we can see that it resembles a ‘maximizing’ signal similar to the one used in project 1. From earlier lectures, we know that a maximizing impulse response would be effective when combined with a signal whose sign matches the flipped version of that impulse response. We can see this effect illustrated in Figure 21

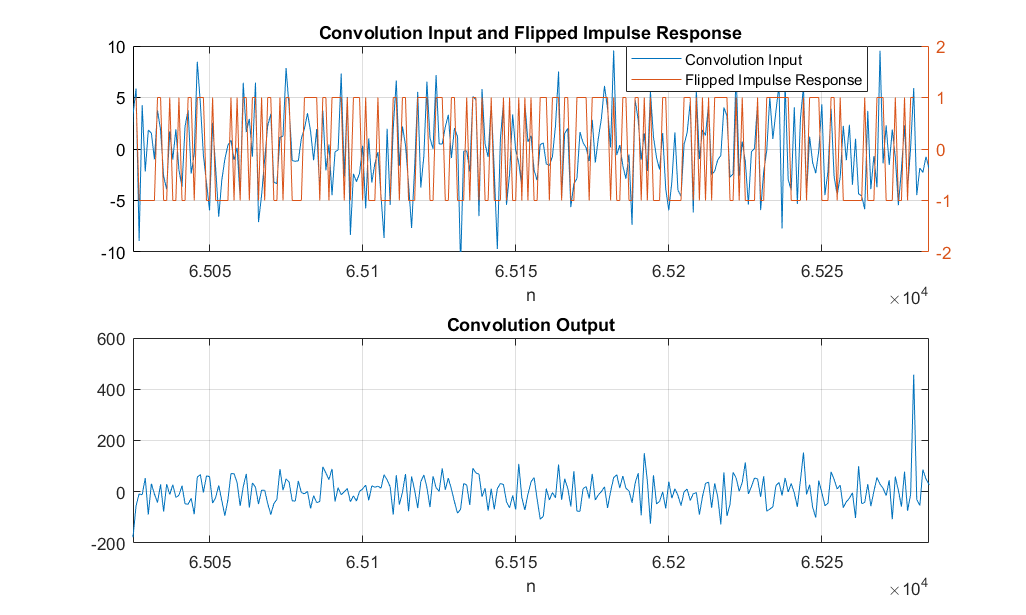


Figure 21 - Flipped impulse response overlaid with the input at the time of the output maximum

From Figure 21, we can see clearly that the sign of the input signal does indeed match up well with the flipped version of the impulse response. We can assume that a similar phenomenon happens for extreme minima in the output except that the input signal matches with the flipped negative version of the impulse response.

**Part 8**

Through inspection of the given SFG, the state space variables can be represented as follows:

|  |
| --- |
|  |
|  |
|  |
| Using the following expression for given in class: |
|  |

My code in p8.m applies the above formula in order to put the above formula into standard form given the state space coefficients.

H\_Numerator =

(c2 + c0\*k2 + c1\*k1 + c2\*k1 + c2\*k2 + c1\*k1\*k2 + c2\*k1\*k2)\*z^2 + (c1 + c0\*k1 + c1\*k2 + c0\*k1\*k2)\*z + c0

H\_Denominator =

z^2 + (k1 + k1\*k2)\*z + k2

Dividing both the numerator and denominator by we have the frequency response in the given form. From this, we can easily represent the and coefficients:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

To validate that this a successful mapping between coefficients, we only need to compare the FUN module output to that of this new SFG. By the power of time travel within this report, we can verify that these functions are valid using the SFG module developed in Part 11:

>> max(abs(y\_fun-y\_p2sfg))

ans =

7.9797e-17

This error value is appropriately small.

**Part 9**

Using similar techniques as in Part 8, (see p9.m) we can express the and coefficients in terms of the and coefficents. The results are shown below.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**Part 10**

In a conventional 2nd order system, the coefficients and dictate the systems stability because they directly correspond to the position of the poles and, more importantly, whether those poles fall inside or outside of the unit circle. We know that and are restricted by the following requirements.

Applying the mapping found in Part 8:

Simplifying:

This BIBO region (if plotted with and as the and axes, respectively) is shaped as square of edge length 2, centered at the origin.

**Part 11**

The implementation of the SFG module can be found in the file proj2sfg.m. One way I can validate the behavior of this module is by also developing a module that uses the state space equations (mentioned in Part 8), found at proj2sfg\_statespace.m. and ensuring their output different is negligible. I also go through the same process using the FUN module developed for the previous project. Provided that the coefficient mappings found in part 8 is correct, I can compare the outputs from these two modules and expect them to have a zero difference. I tested these filters for the coefficients given in Part 13 and I found that my SFG function passed both of these tests.

**Part 12**

To check that this mapping is correct, I simply make sure that this mapping successfully reverses the mapping given in Part 8. There are a few obvious restrictions on the values and , given the denominators of , , and must not equal or ; must not equal .

**Part 13**

*Filter 1*

By finding the roots of the numerator and denominator, we can find the locations of the zeros and poles, respectively. Plotting these complex points as values against the unit circle (shown in Figure XX) gives us an idea of what to expect in the frequency response phase and magnitude. At DC (), the zeros and poles are equidistant because they reflect against the x axis (as a direct result of being complex poles and zeros from a real filter). As increases, the northeast pole-zero pair approaches, the effect of one will generally counter that of the other. Finally, there will be some sort of resonance at about , but since the pole and zero are still roughly equidistant even at this point, we are not likely to see any dramatic changes in magnitude.

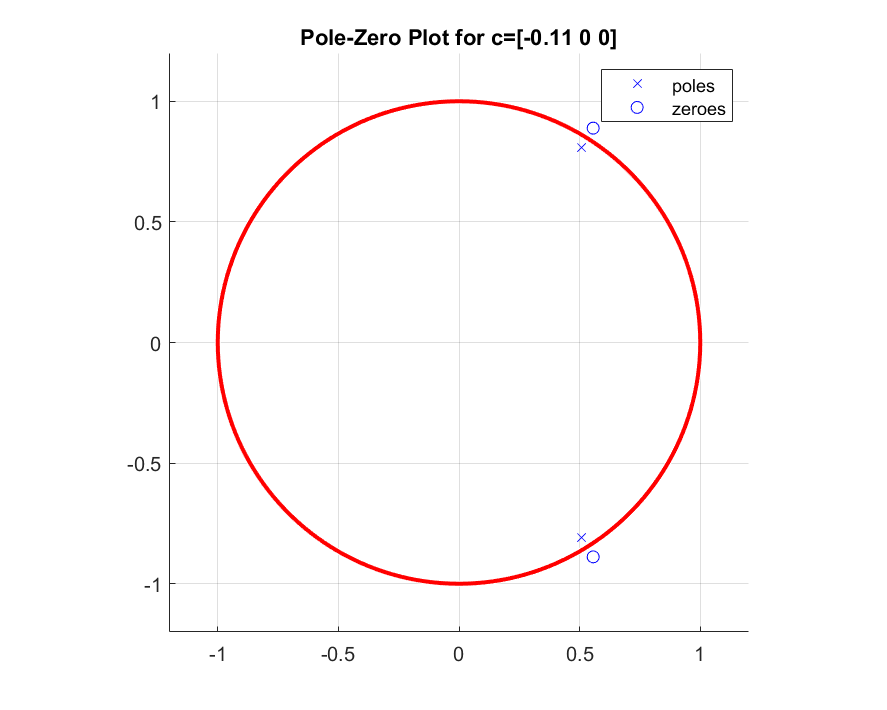


Figure - pole-zero plot for filter 1

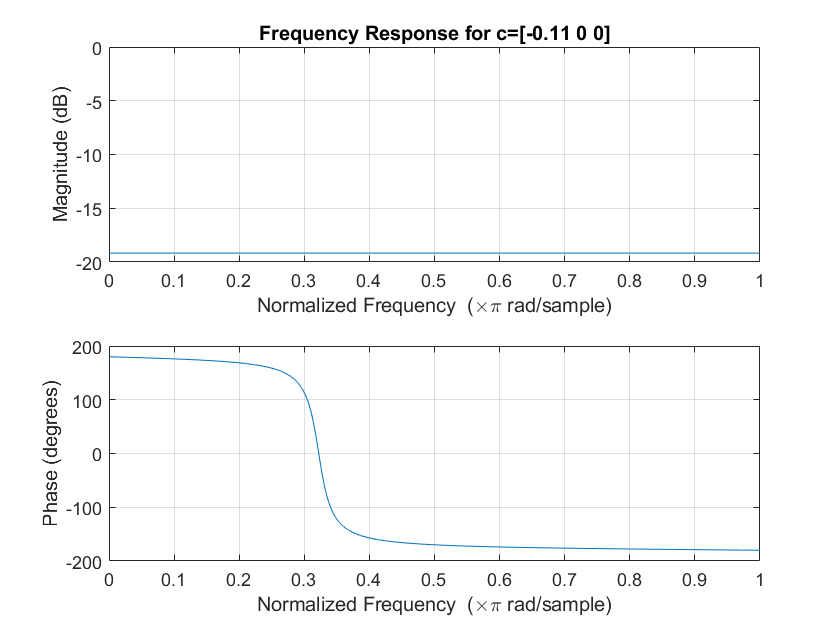
**

Figure - frequency response of filter 1

Figure XX confirms that the frequency response behaves as the poles and zeros would have us expect. It is interesting that the magnitude response stays precisely constant throughout the the entire range of frequencies.

*Filter 2*

We will use the same technique used for filter 1 in order to analyze filter 2. At , we see that there is a zero that falls exactly on the unit circle. As increases, we expect that the frequency response magnitude will increase as we increase the distance between and the pole. Unlike in filter 1, we see a standalone pole at , which will correspond to an unmitigated resonance peak in the magnitude and a rapid change in phase.

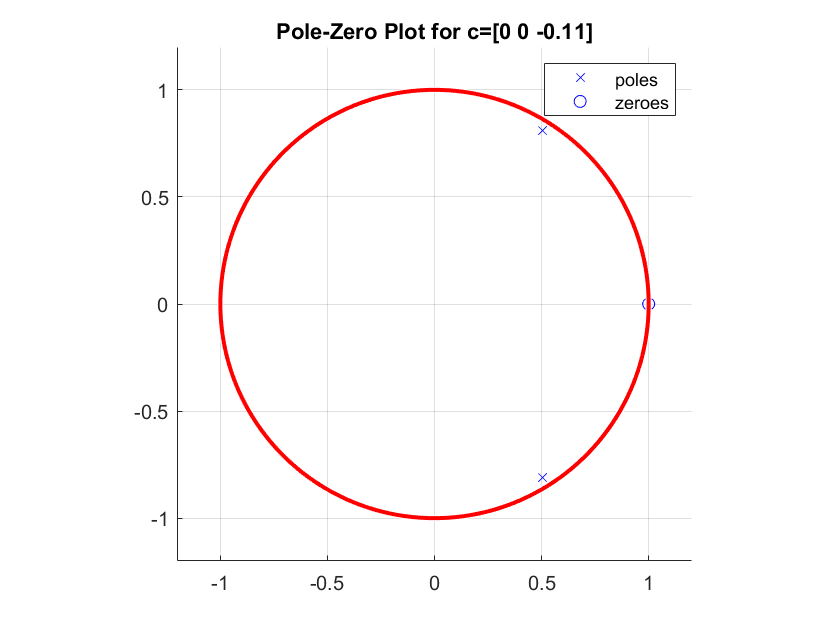
**

Figure - pole-zero plot for filter 2

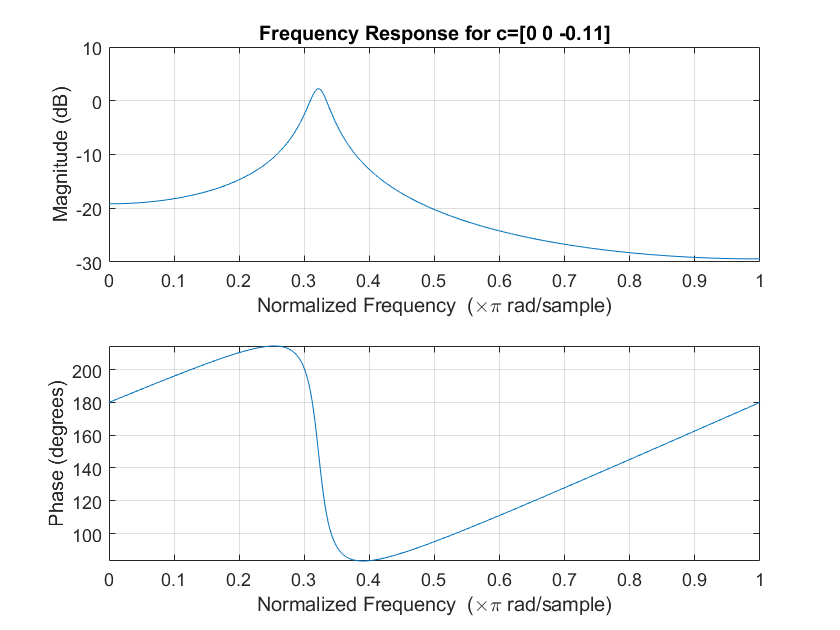
**

Figure - frequency response for filter 2

Figure XX again agrees with the theoretical prediction based off of the positioning of the poles and zeroes.

**Part 14**

*The Problem with Quantization*

Because quantization in a filter presents nonlinear behavior, a quantized filter is liable to produce new frequencies on the output that was not sent to the input. As an example, we can input a sinusoid of fractional frequency equal to .33 to a quantized version of filter 1 in Part 13. Figure XX shows the result using the blackman window to observe the difference between the output of the quantized filter and the output of a linear filter with effectively infinite bit resolution.

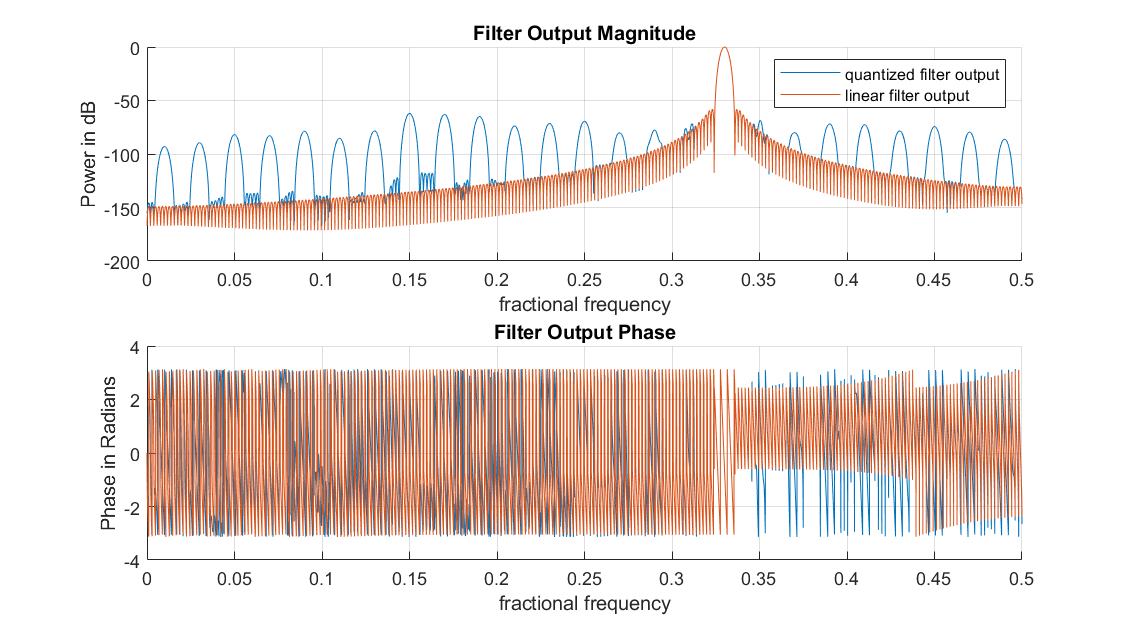
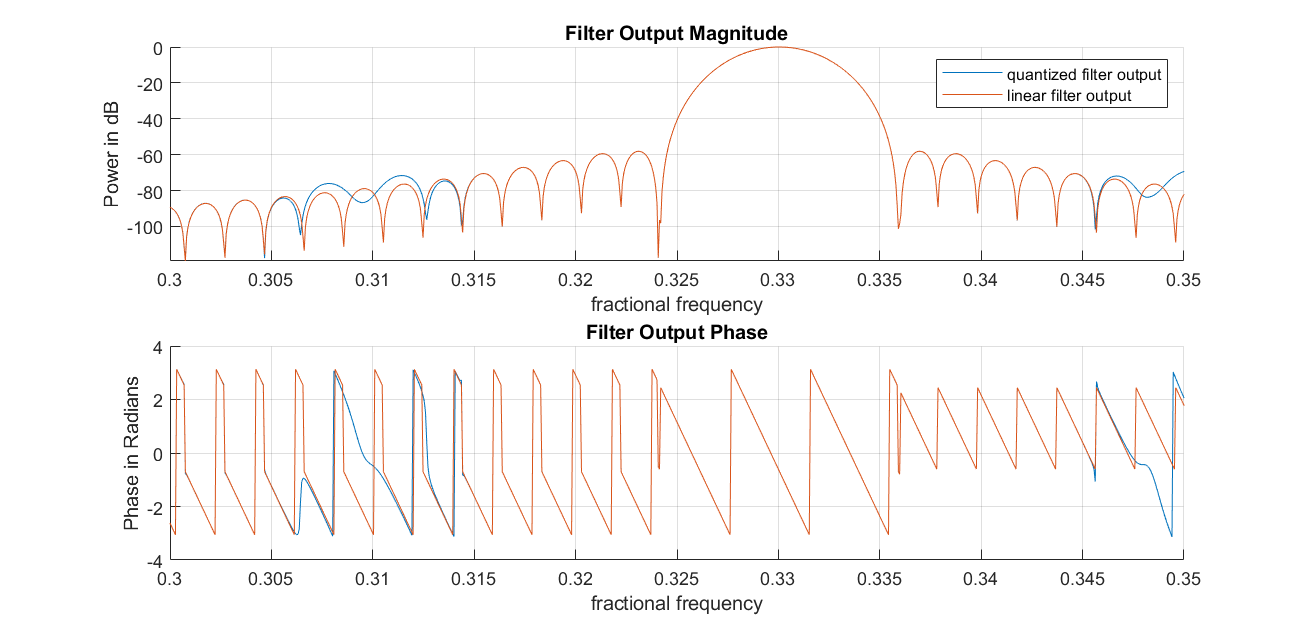


Figure XX shows that there are indeed additional frequencies introduced by the quantized system. In order to find the frequency response of the quantized filter, we should take care to ignore their effect. One way to try to do this is to restrict our window of observation to the frequency of interest. Figure XX shows the a more restricted view.



After focusing in on the fractional frequency of interest, we see that the quantized system varies in its response only minimally at points somewhat far from the main lobe.