**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 distortion of SFDR.

*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

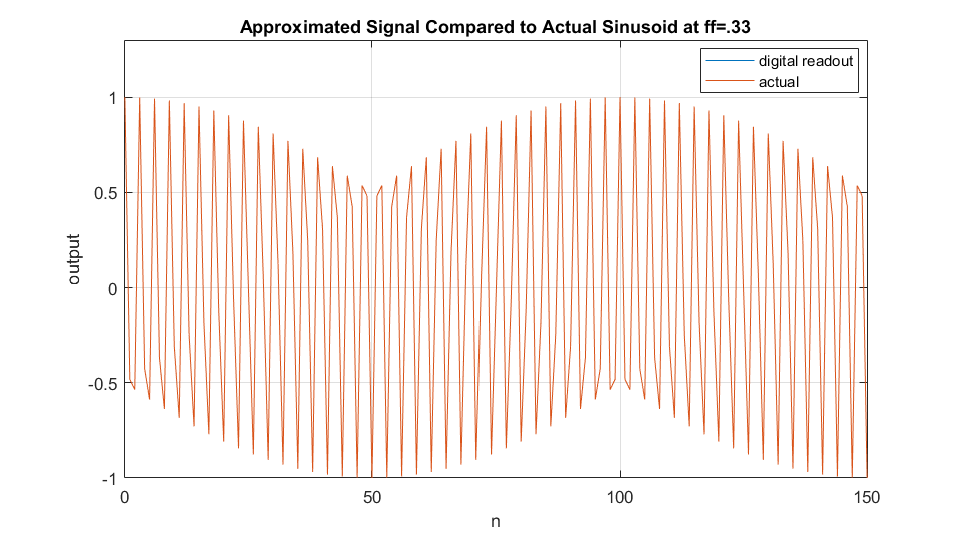


Figure 1 - Time-domain readout compared to the sinusoid it approximates, ff=.33

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.

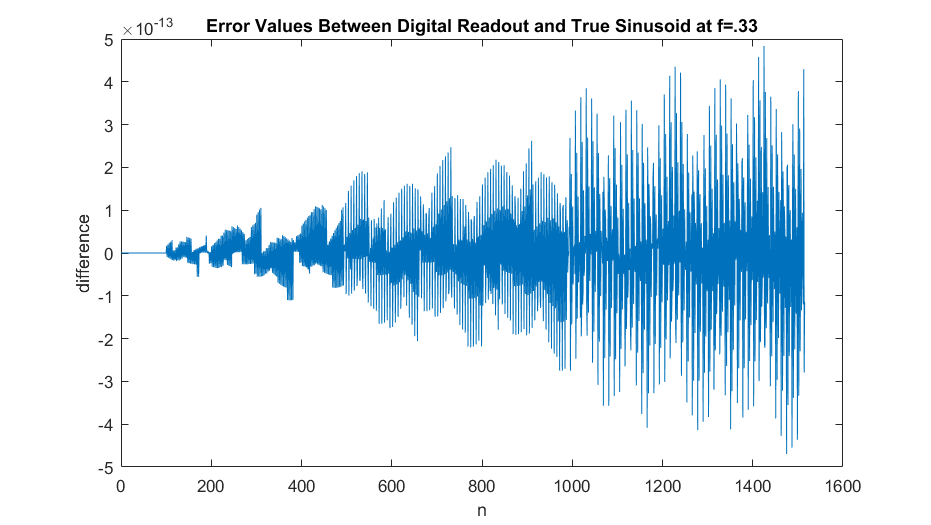


Figure 2 - Digital readout error

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.

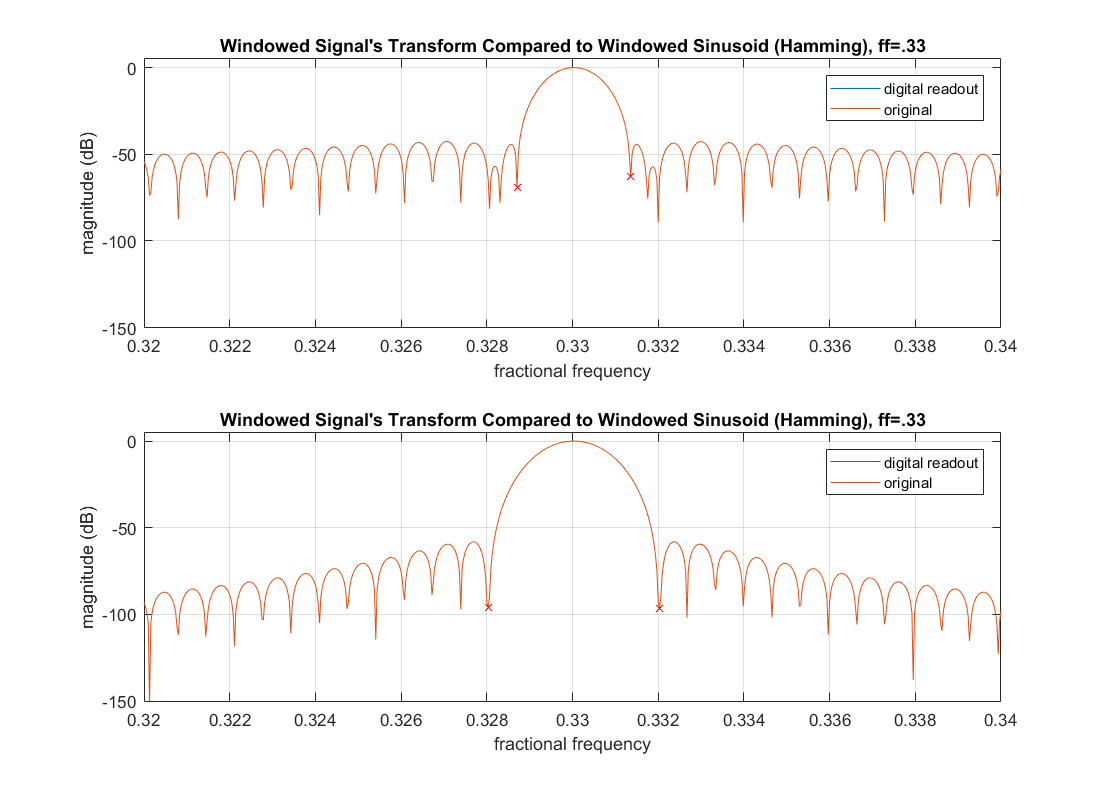


Figure 3 - Comparing the frequency domains using different windows

First, we can make some observations about the window shape 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. The distortion signal calculated using both window types is shown below in Figure 4.

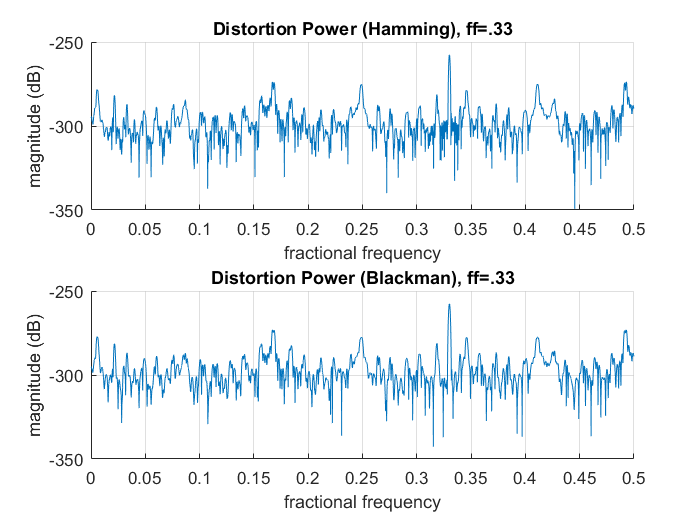


Figure 4 - The frequency content of the distortion signal

Interestingly, the fractional frequency of the sinusoid is frequency at which the distortion power is highest.

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. I will take this power to be my total distortion. Using this method, I get the following result:

**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.

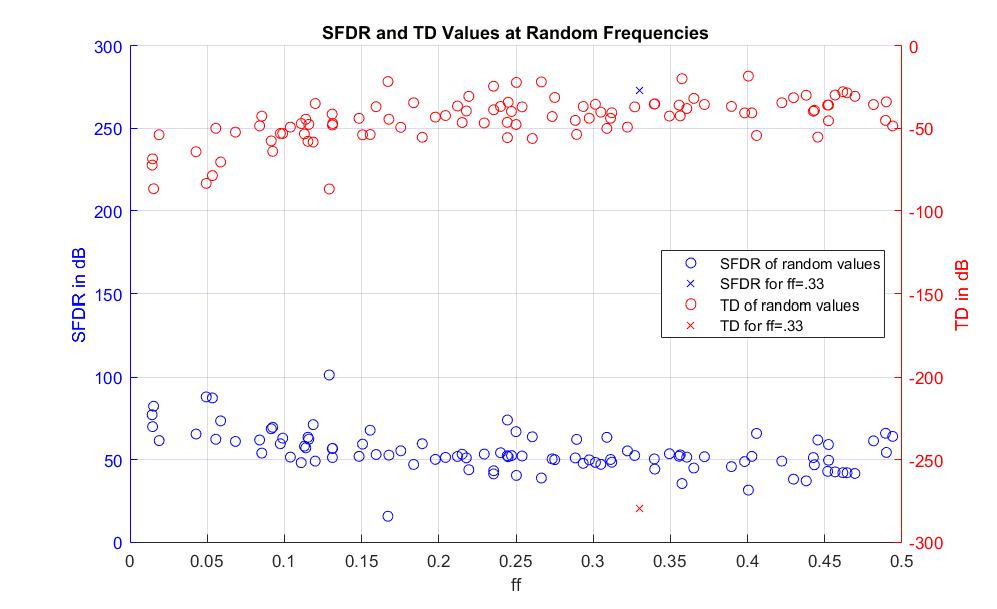


Figure 5 - SFDR and TD of random values plotted over fractional frequency

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.

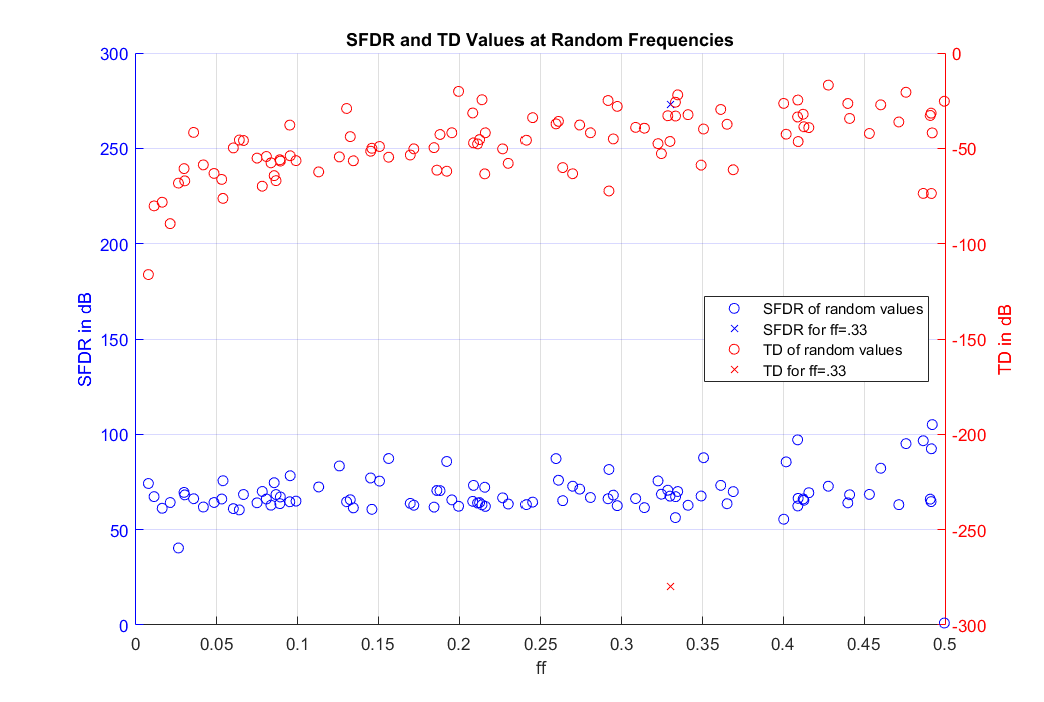


Figure 6 - SFDRs when the lookup table is limited to 1024 samples

As expected, the SFDR has changed from trending down over ff to trending upwards. 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 6 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.

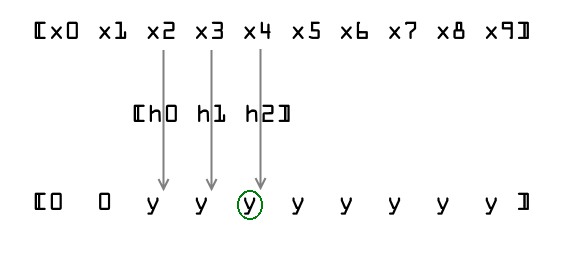


Figure - Illustration of direct convolution with MAC operations illustrated in grey and the final output in green

In Figure 6 we can see that direct convolution requires 3 MACs per output sample. 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.

**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 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 8 shows the result of both direct convolution and my overlap-save algorithm on the first half of the signal. Figure 9 shows the same results for the second signal.

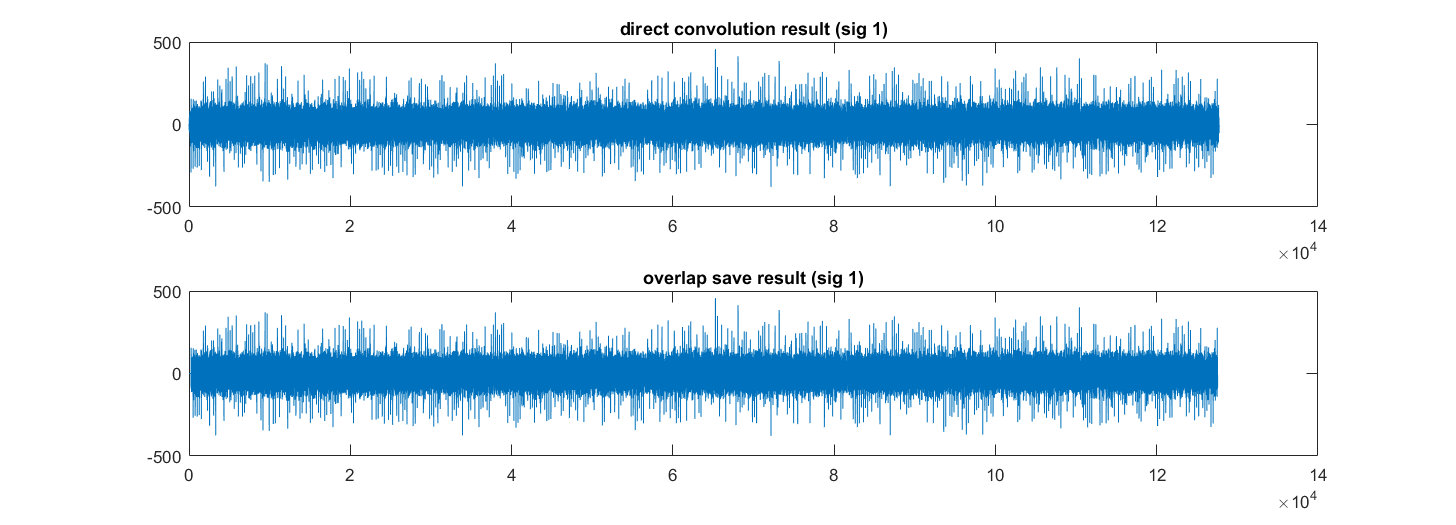


Figure - Comparing output-save results with direct convolution results (signal 1)

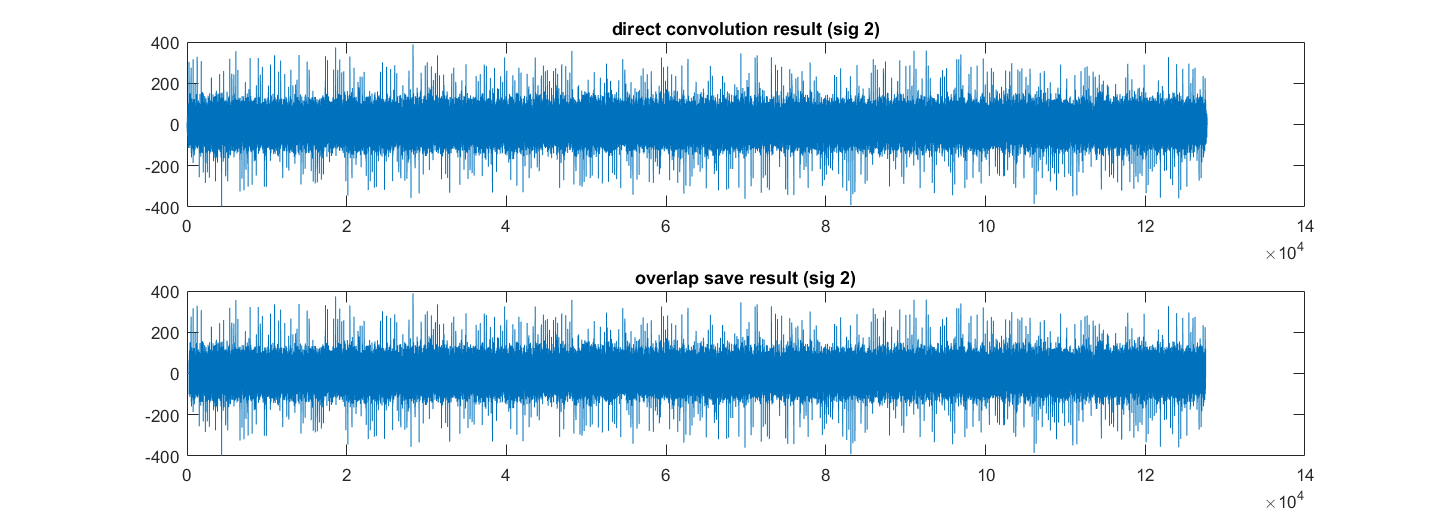


Figure - Comparing output-save results with direct convolution results (signal 2)

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 10 focuses on the differences at the beginning by showing only the first 2000 samples.

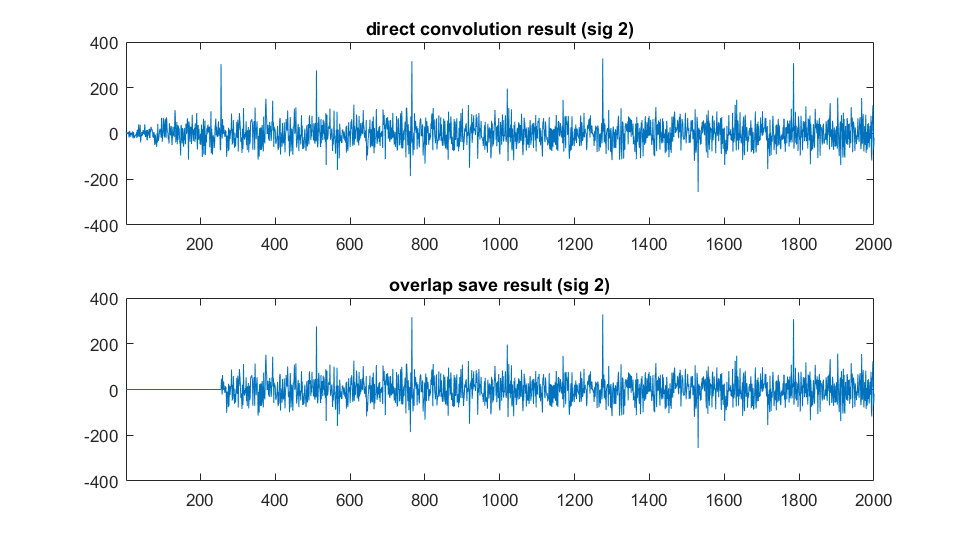


Figure - Differences between the two outputs on 'power-up'

Figure 10 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 11 focuses on the differences at the end by showing only the last 1000 samples.

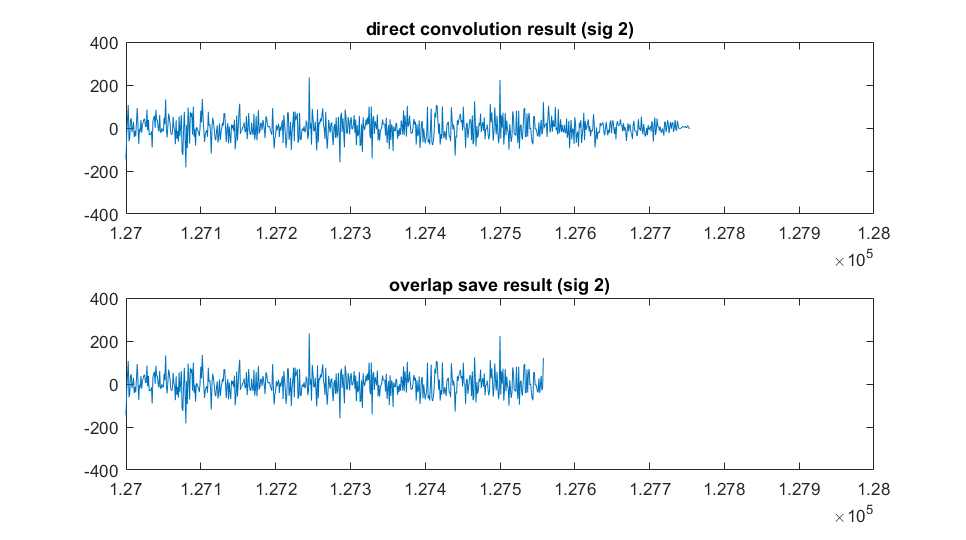


Figure - Differences between the two outputs on 'power-down'

We can see from Figure 11 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 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 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.

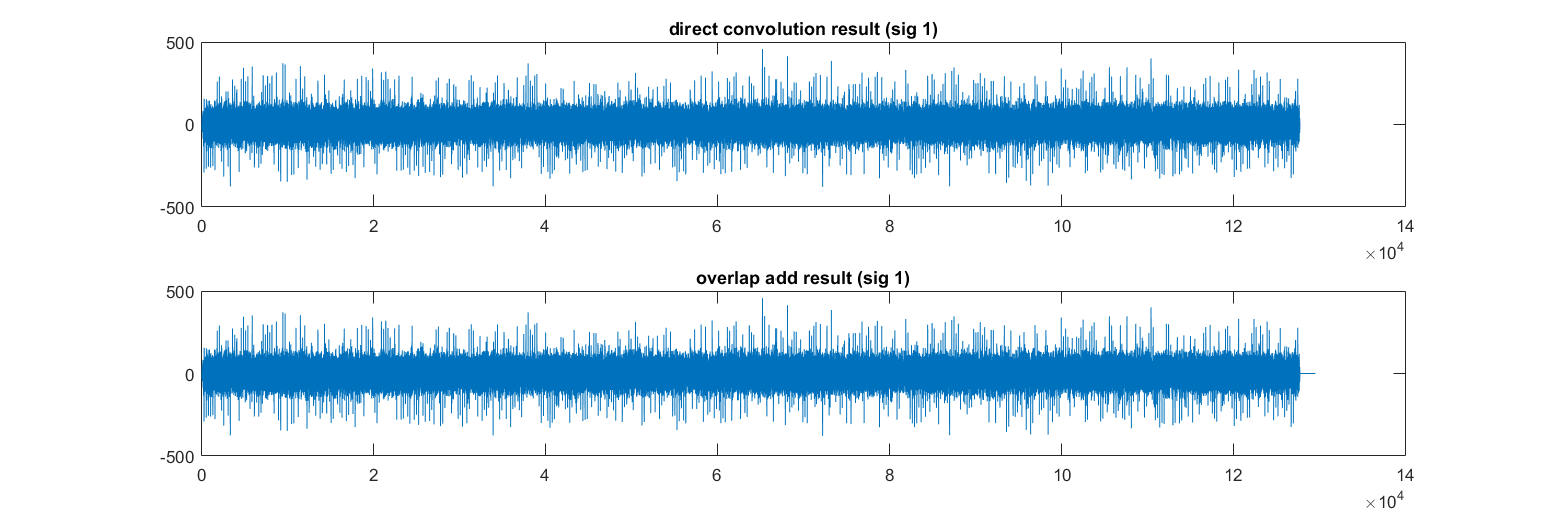


Figure - Comparing output-add results with direct convolution results (signal 1)

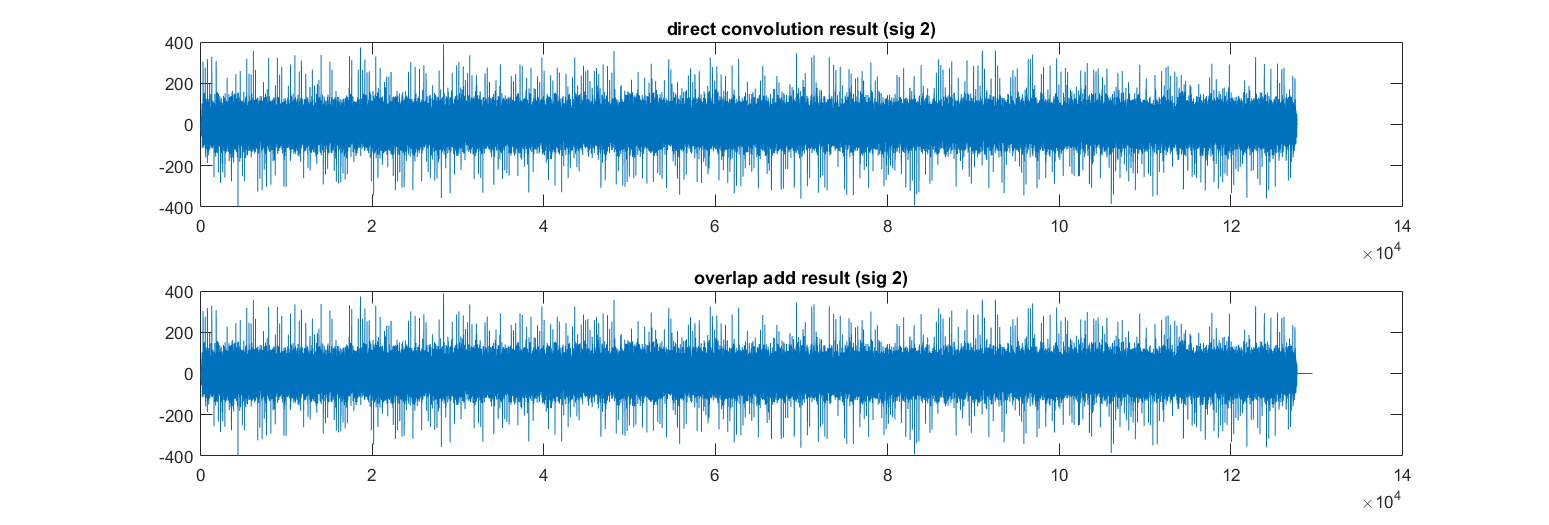


Figure - Comparing output-add results with direct convolution results (signal 2)

The two Figures show that the overlap-save algorithm appears to do a very good job of imitating the behavior of direct convolution. Figure 10 focuses on the two filter results for signal 2 at the beginning by showing only the first 2000 samples.

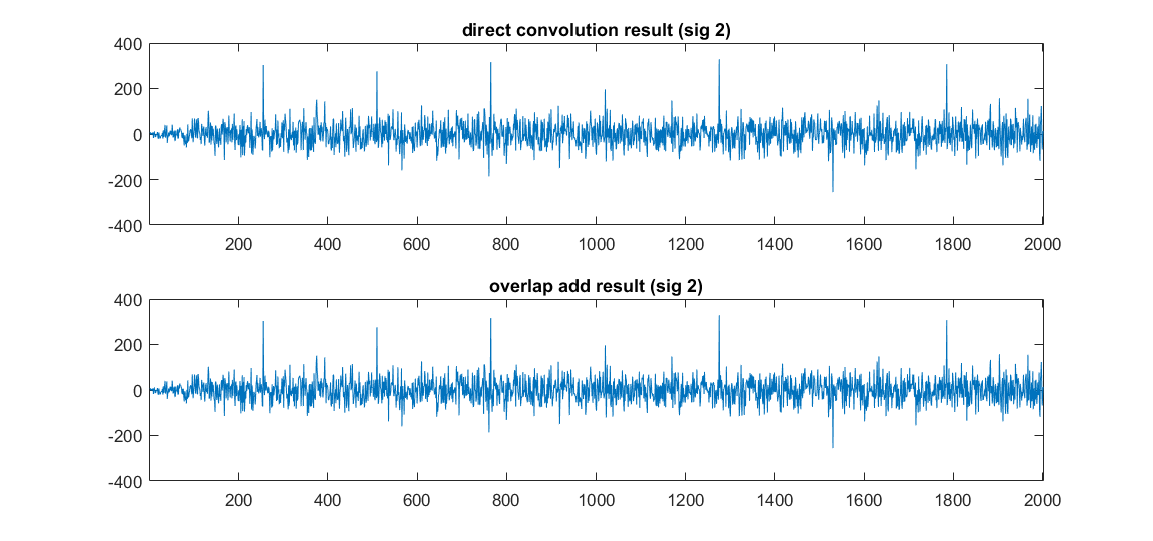


Figure - Differences between the two outputs on 'power-up'

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

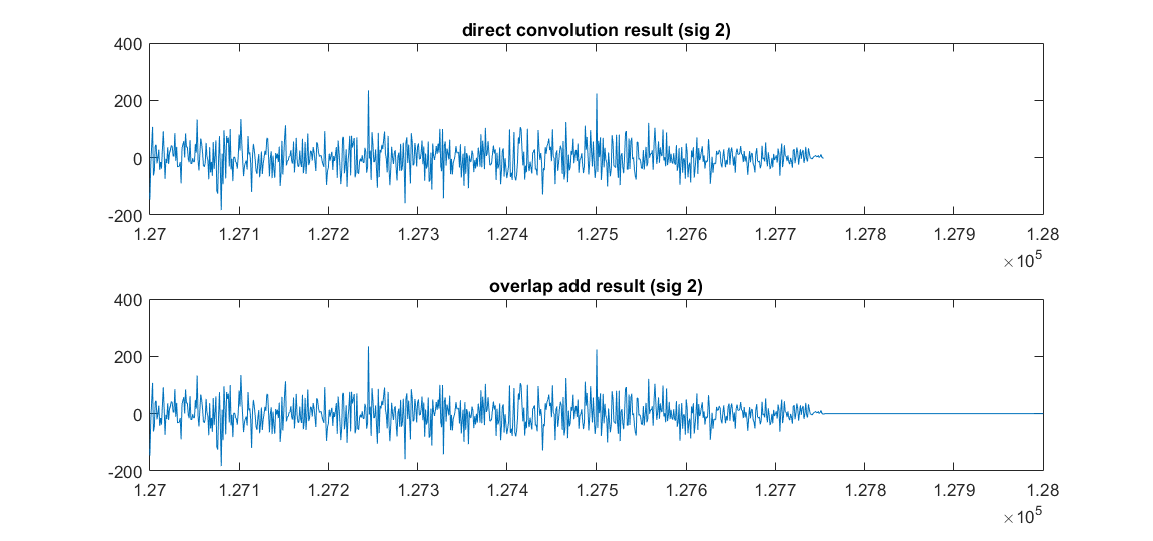


Figure - Differences between the two outputs on 'power-down'

From Figure 15, 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.