CHAPTER 8 REAL-TIME PROCESSING ALGORITHMS

In many applications including digital communications, spectral analysis, audio processing, and radar processing, data is received and must be processed in real-time. In some applications, the incoming data sequence is very large and simply exceeds the memory capacity of the processor. In other applications, the time delay associated with waiting for all the data to arrive before processing is simply unacceptable. Processing data in real-time involves breaking the input data sequence into blocks of data and processing each block of data separately. The chapter begins with the implementation of difference equations in the time domain and the use of circular addressing. Next, Overlap-Add and Overlap-Save, two algorithms for real-time convolution or correlation computations using FFTs, are presented. Block processing of input data for real-time spectral analysis using FFTs is covered in Section 8.3. The Sliding Discrete Fourier Transform, which is a real-time implementation of a Discrete Fourier Transform, is explored in the fourth section. In the final section, adaptive filters for noise and echo cancellation and system identification are introduced.

8.1 DIFFERENCE EQUATIONS AND CIRCULAR ADDRESSING

Difference equations used for FIR filters, IIR filters, and the Goertzel algorithm naturally lend themselves to real-time processing. Consider the difference equation for an FIR filter introduced in Chapter 5:

$$y(k) = \sum_{i=0}^{N} b_i x(k-i) \tag{8.1}$$

The output at any time kT_s is computed by simply multiplying the current input and the N most recent past input values by the filter coefficients and summing up the results. At any sampling time instant, the only data values that need to be stored in memory are the current input and N past inputs. In the next sampling instant, the oldest data value will be discarded and replaced with the newest input data value. Real-time implementation of the filter becomes a matter of taking in a new input value at each sampling instant, discarding the oldest value, then performing the N + 1 multiplications and N additions to calculate the next output. Of course, the processor must be fast enough to perform the calculations within one sampling instant and the input signal itself must be sampled fast enough to maintain fidelity with the original analog signal. IIR filters require storage of a finite number of past outputs in addition to current and past inputs but the basic principle is the same. The Goertzel algorithm, discussed in Chapter 7 for determining the spectrum at a finite number of select frequencies, is simply a 2^{nd} order difference equation requiring two past outputs and the current input.

The input data values and past output values (if needed), can be stored and accessed using circular addressing. The required data is stored in a buffer of fixed length as shown in Figure 8.1. When the new data value comes in, instead of shifting all the data in memory which would use up valuable processing time, the newest value simply replaces the oldest value with a pointer showing where the newest value is located. As multiplications are performed, the pointer moves through the buffer and "circles" or wraps around to the top when the bottom of the buffer is reached.

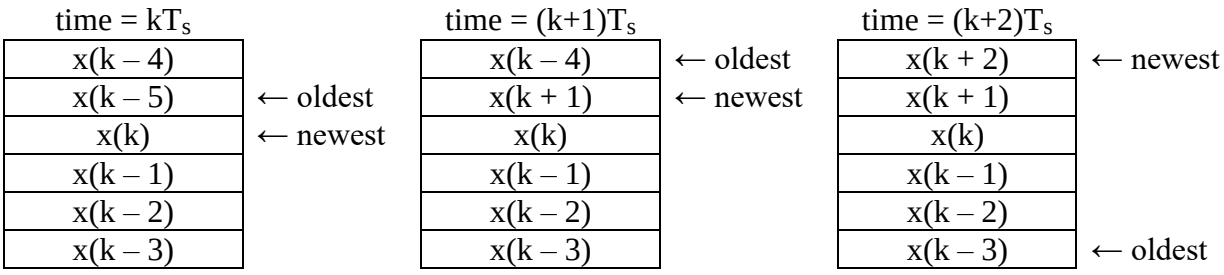


Figure 8.1: Circular Addressing

If the filter order is very large, processing in the time domain is not as efficient as frequency domain processing. As discussed in Chapter 7, FFTs can be used to compute the filter output efficiently for high order filters and long sequences of input data. The next section presents the overlap add and overlap save algorithms for real-time convolution or correlation using blocks of input data and FFTs.

8.2 OVERLAP-ADD AND OVERLAP-SAVE ALGORITHMS FOR CONVOLUTION OR CORRELATION

Overlap-add and overlap-save are useful algorithms for real-time computation of the convolution or correlation of two sequences using FFTs. One practical example of this would be using a digital FIR filter to filter a long sequence of input data. In this case, the filter output is the convolution of the input data and the filter coefficients. Rather than waiting for all the input data to arrive, the convolution can be processed in blocks. Recall from Chapter 7 that if the filter has an order $N \geq 128$, it is computationally efficient to use FFTs to compute the convolution (fast convolution) instead of working in the time domain. Another practical application would be to correlate an incoming stream of data with a template of reference signals for voice identification, finger print identification, radar target identification, or other purposes. The correlation can be computed in real-time using FFTs as blocks of input data become available.

Overlap-Add Algorithm

Assume the data input sequence, x(k) has length L, the filter impulse response, h(k), has length N, and L >> N). The procedure to compute y(k) = x(k)*h(k) is as follows:

1. Partition x(k) into blocks of length N by simply using data buffers of length N. Call these partitioned sequences x_0, x_1, x_2, …

2. Convolve each block of input data, as it becomes available, with h(k) using fast convolution (FFTs):

$$y_i = \text{ifft}\left[\text{fft}\left(x_i^{\text{padded}}\right) \cdot \text{fft}\left(h_i^{\text{padded}}\right)\right]$$

   Note that the block of input data and the filter coefficients must be padded with zeros so each have length 2N – 1. The result of each block convolution will be an output sequence also of length 2N – 1. Call these blocks of results y_0, y_1, y_2, …

3. The output sequence is then constructed from the individual convolution results by

$$y(k) = y_o(k) + y_1(k-N) + y_2(k-2N) + \ldots$$

The over-lap add algorithm is illustrated in Example 8.1 using very simple and short sequences.

Example 8.1: Computing Convolution using Overlap Add

Suppose x(k) = [–1 4 2 5 1 3 7] and h(k) = [1 3]. Compute y(k) = x(k) * h(k) using the overlap add algorithm and check the results using the MATLAB® function, ***conv***.

Solution

1. First partition x(k) into sequences of length N = 2:

   x0 = [–1 4] x1 = [2 5] x2 = [1 3] x4 = [7 0]

2. Now compute each individual block convolution (since this example is being done by hand with very short sequences, the convolution will be computed in the time domain rather than using FFTs):

   y0 = x0 * h(k) = [–1 1 12]
   y1 = x1 * h(k) = [2 11 15]
   y2 = x2 * h(k) = [1 6 9]
   y3 = x3 * h(k) = [7 21 0]

3. Now construct y(k) from the individual block convolutions:

```
          [ –1   1   12 ]
                [ 2   11   15]
                      [ 1    6    9]
                            [ 7   21   0]
y(k) =   [ –1   1   14   11   16   6   16   21   0]
```

Checking the result with MATLAB yields the same result:

```
x = [-1 4 2 5 1 3 7]; h = [1 3]; y = conv(x,h)

y =    -1     1    14    11    16     6    16    21
```

Overlap-Save Algorithm

Assume the data input sequence, x(k) has length L, the filter impulse response, h(k), has length N, and L >> N). The procedure to compute y(k) = x(k)*h(k) is as follows:

1. Add N – 1 leading zeros to the input data sequence, x(k).
2. Partition the input data sequence (with the added zeros) into segments that overlap by N – 1. Choose a segment length M ≈ 2N which is preferably a power of 2. Call these partitioned sequences x_0, x_1, x_2, …
3. Zero pad h(k) using trailing zeros so that it is length M.
4. Perform *periodic* convolutions on the blocks of data (as they become available in real time) and the padded h(k) using FFTs. This will yield sequences y_0, y_1, y_2, … of length M:

$$y_i = ifft\,[fft(x_i) \cdot fft(h_i^{padded})]$$

5. Discard the first N – 1 samples from each of the individual convolutions, then paste the resulting segments together to form the output, y(k).

Example 8.2: Computing Convolution using Overlap Save

Suppose x(k) = [–1 4 2 5 1 3 7] and h(k) = [1 3]. Compute y(k) = x(k) * h(k) using the overlap save algorithm and compare the results to Example 8.1.

Solution

In this example, L = 7 and N = 2.

1. First, add N – 1 = 1 leading zero to the input sequence:

   x = [0 –1 4 2 5 1 3 7]

2. Next, partition x into segments of length M = 4 (≈2N and a power of 2) that overlap by N – 1 = 1:

   x0 = [0 –1 4 2] x1 = [2 5 1 3] x2 = [3 7 0 0]

3. Pad h(k) with trailing zeros to make it length M = 4:

   h = [1 3 0 0]

4. Convolve the sections using periodic convolution (regular convolution with wraparound to make length = 4). Once again, since the sequences are so short for this example, computations will be done in the time domain instead of using FFTs:

   x0 * h = [0 –1 1 14 6 0 0] → y0 = x0 ● h = [6 –1 1 14]

   x1 * h = [2 11 16 6 9 0 0] → y1 = x1 ● h = [11 11 16 6]

$$x2 * h = [\ 3\ \ 16\ \ 21\ \ 0\ \ \underline{0\ \ 0\ \ 0}\ ] \ \rightarrow\ y2 = x2 \bullet h = [\ 3\ \ 16\ \ 21\ \ 0]$$

5. Discard the first N – 1 = 1 samples from each section then past directly together to get the output:

$$y(k) = [\ -1\ \ 1\ \ 14\ \ 11\ \ 16\ \ 6\ \ 16\ \ 21\ \ 0]$$

The results of this example are identical to the results in example 8.1.

Overlap add and overlap save are both efficient algorithms for computing a convolution or correlation in real-time. How do the two algorithms compare?

- The overlap save algorithm does not require that any of the previous convolution results be saved since the output is determined simply by pasting individual sections together. However, it does require that some of the input data from the previous partition be saved and used in the next data segment and convolution calculation.
- The overlap add algorithm requires that one previous convolution result be saved and combined with the current convolution result. However, once a block of input data is used for convolution, it can be discarded.
- The convolution step in both overlap add and overlap save uses FFTs (fast convolution). Overlap save is set up beautifully for fast convolution since the segment convolutions are periodic convolutions. Overlap add requires zero padding of the input segments and filter coefficients following the procedure outlined in Chapter 7 on fast convolution.
- For both methods, the FFT of the padded filter coefficients, h(k), can be computed ahead of time and stored in memory.
- Both methods have about the same computational complexity – the number of real multiplications is roughly

$$\frac{\mathrm{L}}{\mathrm{M}} \cdot \mathrm{M}^2 \quad \text{using time domain convolution}$$

$$L/M\ [12\bar{\mathrm{N}}\log_2(2\bar{\mathrm{N}}) + 8\bar{\mathrm{N}}] \quad \text{using FFTs for the convolution} \tag{8.2}$$

where L is the length of the input data sequence, x(k), N is the length of h(k), M is the length of the partitions of x(k), and $\bar{N}$ is the smallest power of 2 that is greater than or equal to M. Compare this equation to the number of multiplies for fast convolution summarized in Section 7.5. The only difference is the pre-multiplier of L/M which is simply the number of partitions of the input sequence, x(k), or equivalently the number of individual convolutions that need to be performed.

8.3 REAL-TIME SPECTRAL ANALYSIS USING FAST FOURIER TRANSFORMS

Analyzing the spectrum of a signal in real-time using Fast Fourier Transforms is accomplished by collecting blocks of input data samples and performing an FFT on each block of data. The process is illustrated in Figure 8.2.

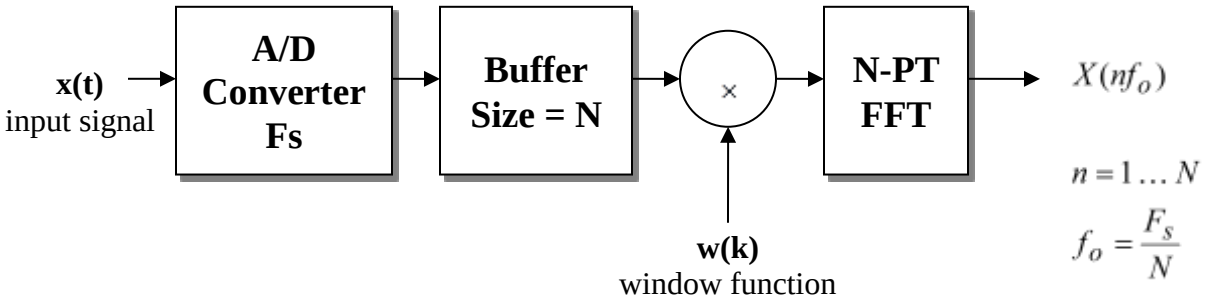


Figure 8.2: Real-Time FFT Processing

As indicated in Figure 8.2, the input signal is sampled and these samples are stored in a buffer of size N. Once the buffer is full, an FFT is computed for the block of N data samples. As discussed in Section 7.2, the input data samples are windowed prior to computing the N-PT FFT to enhance the spectral analysis.

There are several important considerations in performing real-time FFT processing:

- The sampling rate, Fs, must of course be chosen to exceed twice the bandwidth of the input signal to avoid aliasing.
- The duration of time, D, over which the input signal is sampled for each FFT computation must be chosen carefully. Increasing D will improve the resolution of the FFT since $f_o = 1/D$. However, if the time duration is chosen too long, then changes in the spectrum of input signal could be missed. The time duration, D, directly affects the size of the FFT since $N = D \cdot Fs$ which means a longer time duration results in a larger FFT.
- The choice of a window function is application dependent. As discussed in Section 7.2, windowing can be used to reduce the effects of leakage. Each window function has its own somewhat unique characteristics which affect how the leakage is re-distributed.
- Overlapping the blocks of input data can be very beneficial to the spectral analysis, particularly when windowing is included. In other words, if the block size is 1024, the last 256 input samples of a block could become the first 256 samples of the next block.

As mentioned above, the choice of time duration for each block of input data has a big effect on the resulting FFT. If the time duration is too long, then short burst changes in the input signal could be missed (poor time resolution). Decreasing the time duration, D, will certainly improve the time resolution but will definitely degrade the frequency resolution. Example 8.3 illustrates this trade-off between time resolution and frequency resolution.

Example 8.3: Trade-off Between Time Resolution and Frequency Resolution

(a) Suppose the input signal is $x_1(t) = \begin{cases} \sin(2\pi(100)t) & 0s \le t < 0.75s \\ \sin(2\pi(300)t) & 0.75s \le t < 1.25s \\ \sin(2\pi(100)t) & 1.25s \le t \le 2.0s \end{cases}$

The input signal is sampled at Fs = 1000 Hz (plenty fast) and the buffer length (FFT length) is chosen to be N = 256. The time duration, D, would therefore be 0.256 sec and the frequency resolution would be 1000/256 ≈ 4 Hz. Using Simulink® and MATLAB, plot the FFT of $x_1(t)$ using block processing.

(b) Now change the input signal to $x_2(t) = \begin{cases} \sin(2\pi(100)t) & 0s \le t < 0.95s \\ \sin(2\pi(300)t) & 0.95s \le t < 1.05s \\ \sin(2\pi(100)t) & 1.05s \le t \le 2.0s \end{cases}$

Use the same sampling frequency and FFT length from part (a), plot the FFT of $x_2(t)$ using block processing, and comment on the results.

(c) Repeat part (b) with the FFT length reduced to 128. Reducing the FFT length from 256 to 128 will cut the time duration in half (D = 0.128 sec) and double the frequency resolution (fo = 1000/128 = 7.8125 Hz).

(d) Repeat part (b) with the FFT length reduced to 64. In this case, the time duration is further reduced to 0.064 sec while the frequency resolution increases to 15.625 Hz.

Solution

A Simulink model (Figure 8.3) is built for the block processing of the input signal and FFT computation.

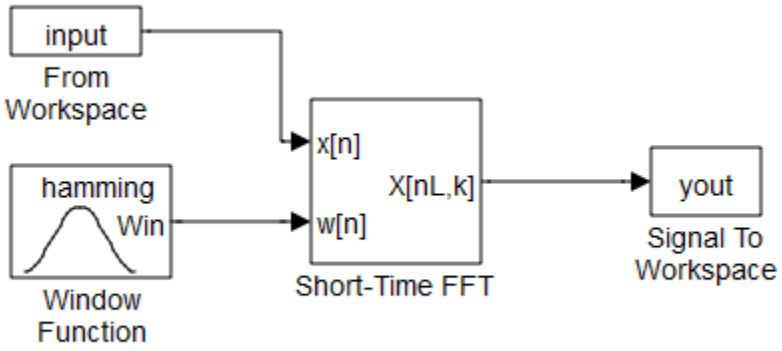


Figure 8.3: Simulink Model for Block Processing of FFT

In the Window Function block, the window length is set equal to the FFT length, sample mode is Discrete, and sample time is 1/Fs. In the Short-Time FFT block, the analysis window length is set equal to the FFT length and the overlap is set to 0. The sampling frequency is specified in the From Workspace block.

(a) The following commands in MATLAB define the input signal then plot the results of running the Simulink model

```
Fs=1000; N = 256;
t=0:1/Fs:2;      % Sampling times
f=-N/2*Fs/N:Fs/N:(N-1)/2*Fs/N;  % Frequency bins for FFT
% Define input signal, x, to be a 300 Hz sine wave from 0.75 to
% 1.25 sec and 100 Hz at all other times
x=sin(2*pi*100*t);
x(750:1250)=sin(2*pi*300*t(750:1250));
input=[t' x'];
[K,F]=meshgrid(t,f);  % Set up mesh grid for 3-D plot
```

Now run FFT Block Simulation (stop time should be set to 2 sec) then execute the following commands in the MATLAB workspace

```
yfft(:,:)=yout(:,1,:);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (a):

The resulting FFT spectrum is shown in Figure 8.4 (a). It is clear from the plot that the input initially has a frequency component close to 100 Hz, the spectrum then switches to approximately 300 Hz, and then returns to 100 Hz. Note that there is a time delay of approximately 0.25 sec in the spectrum. This is the length of time required to fill the buffer with 256 input values before the processing can begin.

Challenge Question 8.1

Close examination of the plot in Figure 8.4(a) reveals a very small peak at 300 Hz prior to the much larger peak and a very small peak at 100 Hz in between the two much larger peaks. Why does this occur?

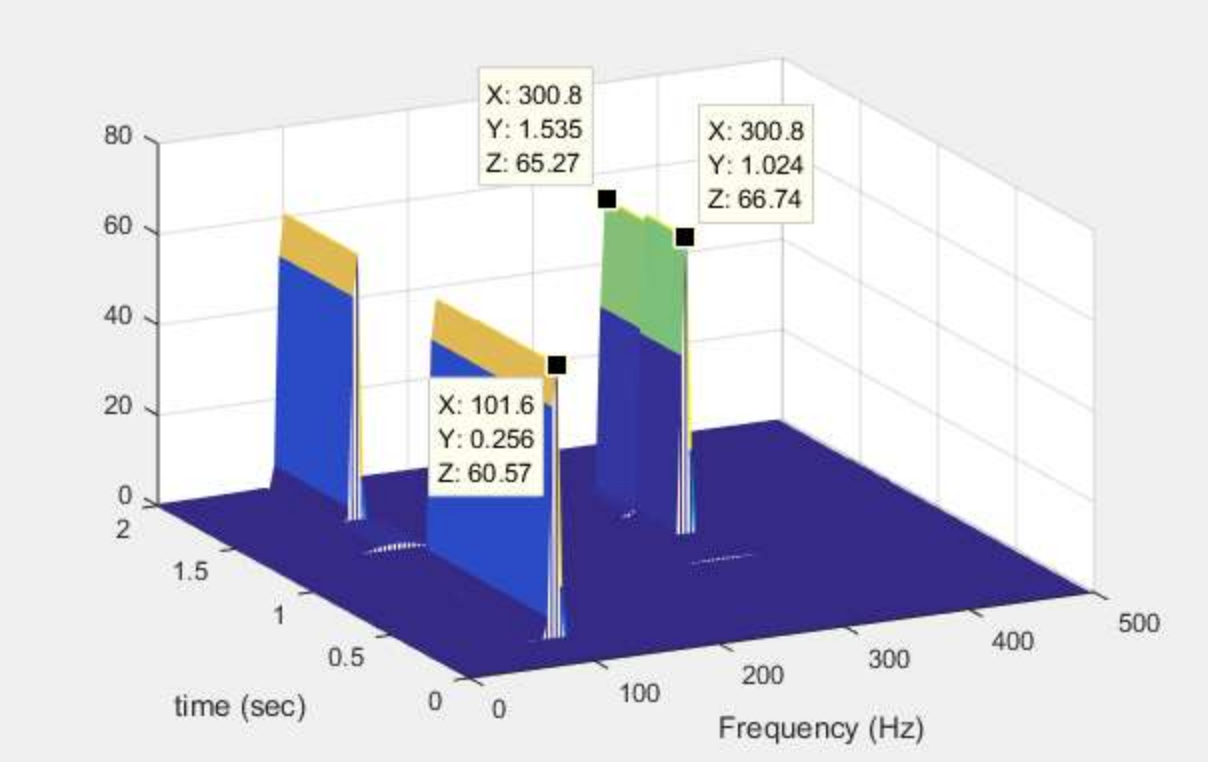


Figure 8.4a: Spectrum for Example 8.3(a)

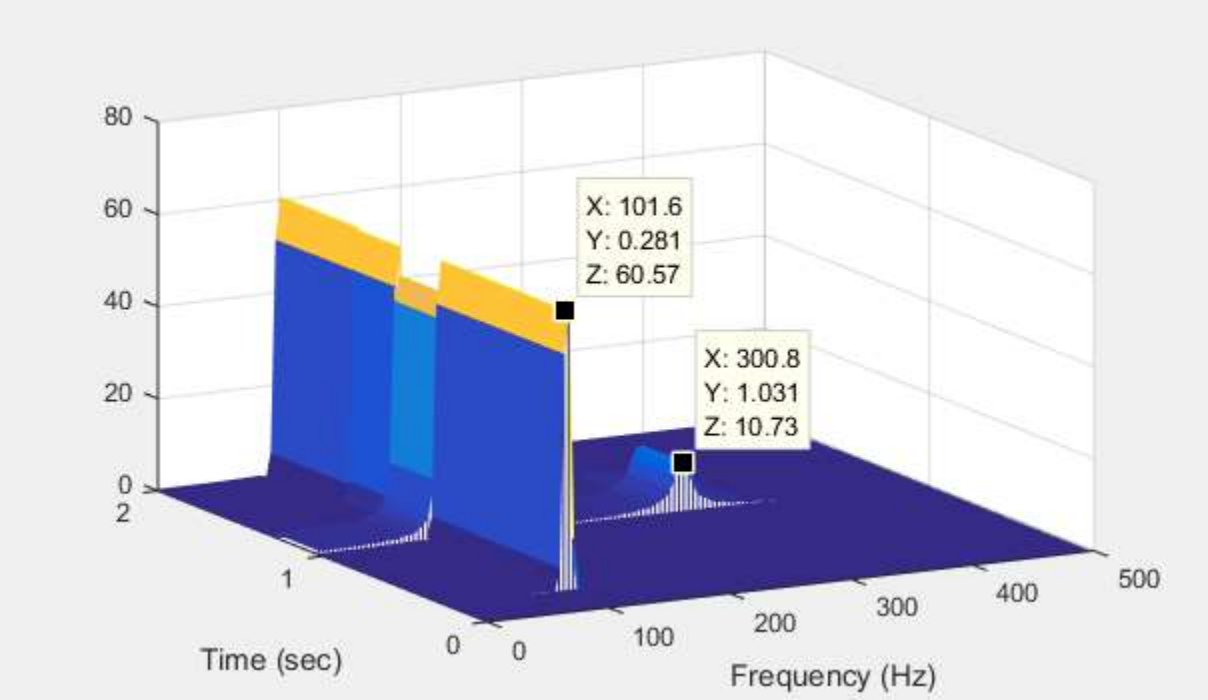


Figure 8.4b: Spectrum for Example 8.3(b)

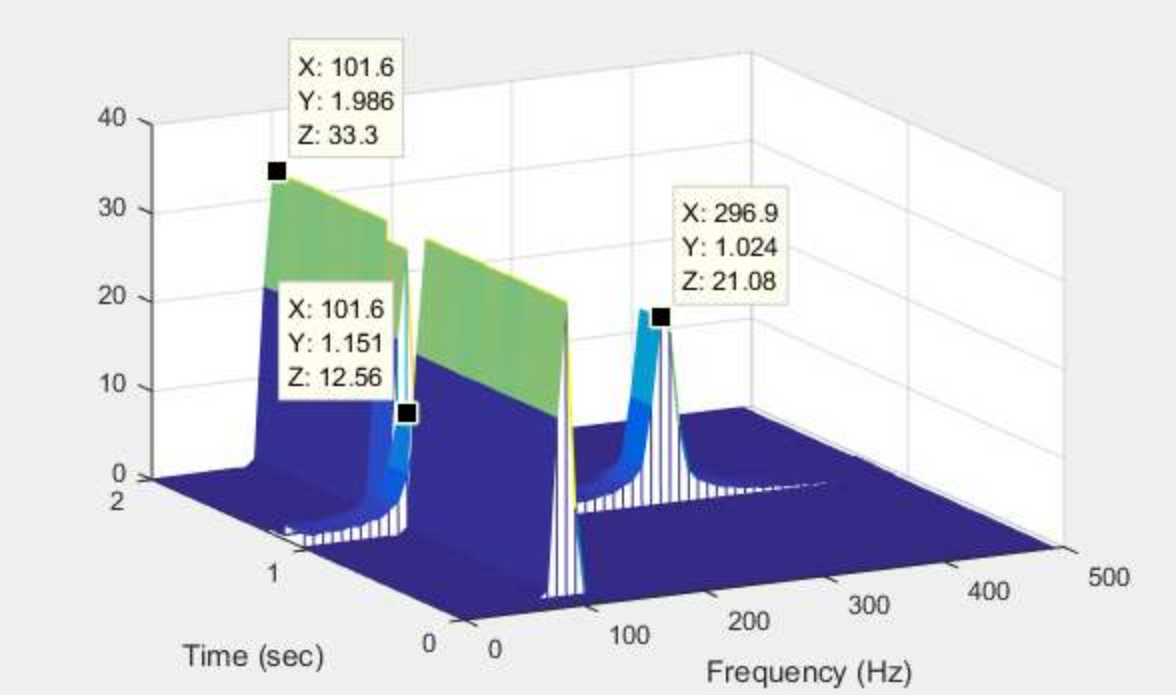


Figure 8.4c: Spectrum for Example 8.3(c)

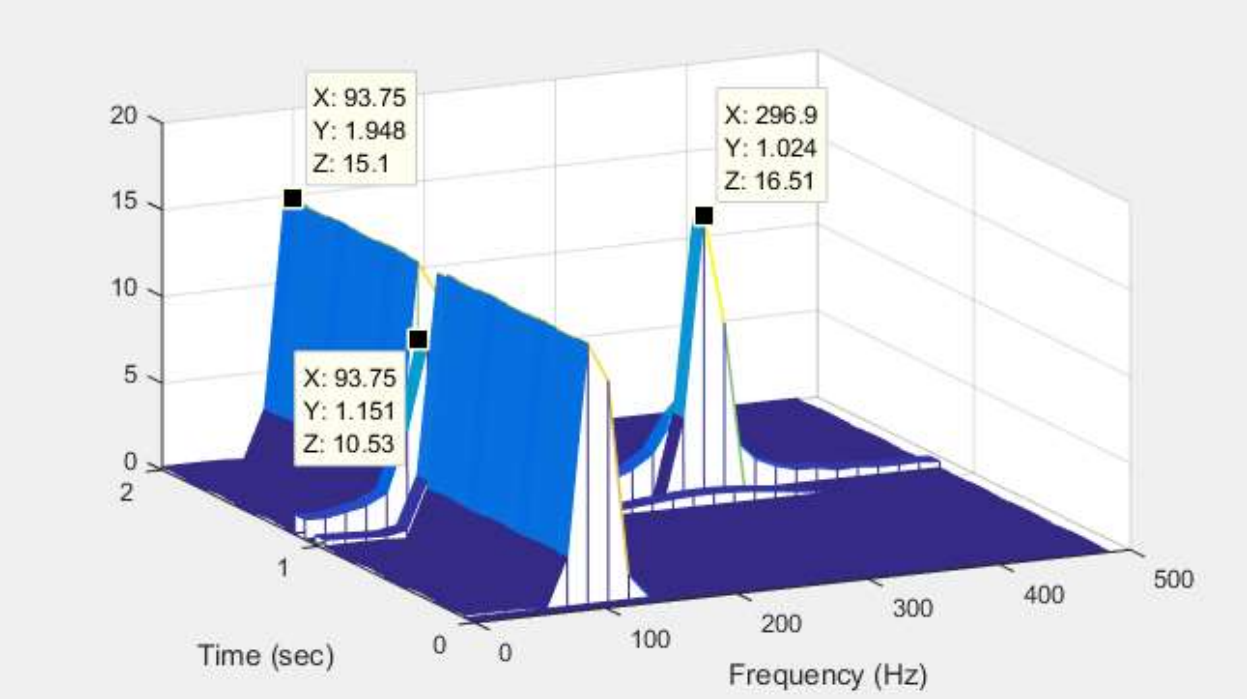


Figure 8.4d: Spectrum for Example 8.3(d)

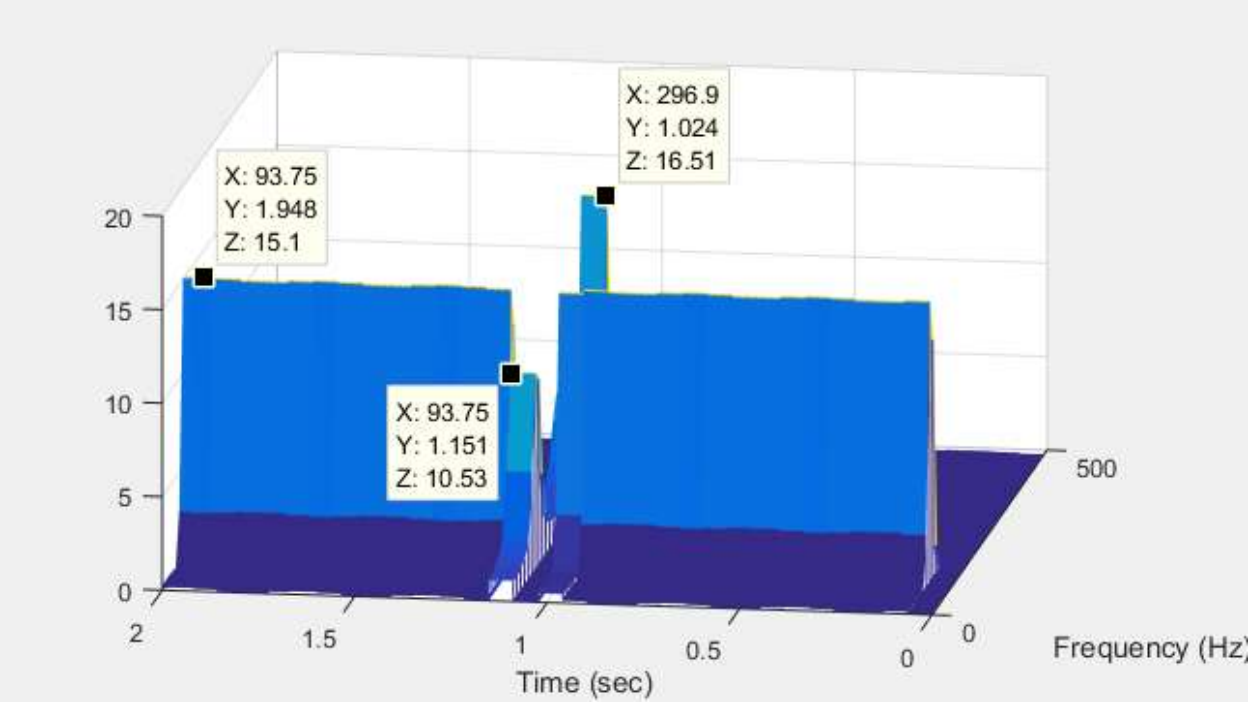


Figure 8.4e: Spectrum for Part (d) – Rotated for a Different Perspective

Figure 8.4: Spectral Plots for Example 8.3

(b) The following commands in MATLAB define the input signal then plot the results of running the Simulink model

```
x=sin(2*pi*100*t);
x(950:1050)=sin(2*pi*300*t(950:1050));
input=[t' x'];
```

Re- run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:,:)=yout(:,1,:);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (b):

The resulting FFT spectrum is shown in Figure 8.4(b). Notice that a small peak at 300 Hz does appear in the spectrum; however, during the exact same period of time there is a much larger peak occurring at approximately 100 Hz. The fact that the input signal is a pure 300 Hz wave between 0.95 sec and 1.05 sec is completely lost because the duration of time, D, over which blocks of input data are being analyzed is much longer than the period of time during which the input signal hits 300 Hz. In other words, the buffer of input signal samples does include samples when the input is a 300 Hz tone, but it also includes a lot more samples of the 100 Hz tone. Parts (c) and (d) of this example are an attempt to fix this problem by reducing the length of time over which the input signal is sampled.

(c) The following commands in MATLAB adjust the buffer length to 128

```
% Change N to 128 and adjust frequency bins and meshgrid:
Fs=1000; N = 128;
f=-N/2*Fs/N:Fs/N:(N-1)/2*Fs/N; % Frequency bins for FFT
[K,F]=meshgrid(t,f); % Set up mesh grid for 3-D plot
```

Change the window lengths and FFT length in FFT Block Simulation to 128. Re-run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:,:)=yout(:,1,:);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (c):
The resulting FFT spectrum is shown in Figure 8.4(c). The 300 Hz signal is much more distinct in this spectrum; however, the spectrum still shows a 100 Hz signal (albeit much smaller) occurring in the same time period. Also notice the widening of the peaks due to poorer frequency resolution.

(d) The following commands in MATLAB adjust the buffer length to 64

```
%Change N to 64 and adjust frequency bins and meshgrid:
Fs=1000; N = 64;
f=-N/2*Fs/N:Fs/N:(N-1)/2*Fs/N; % Frequency bins for FFT
[K,F]=meshgrid(t,f); % Set up mesh grid for 3-D plot
```

Change the window lengths and FFT length in FFT Block Simulation to 64. Re-run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:,:)=yout(:,1,:);
mesh(F,K,abs(fftshift(yfft,1)))
```

Comments on Part (d):
The resulting FFT spectrum is shown in Figure 8.4(d) and (e). The time resolution has been improved enough to show the 300 Hz component occurring alone for a small period of time. Notice the degradation in the frequency resolution. There is now much more uncertainty in the actual frequency of the two tones.

Example 8.3 illustrates the effect of time duration on a real-time FFT. The next example shows that overlapping the blocks of input data can be beneficial for real-time spectral analysis.

Example 8.4: Overlapping Blocks of Data for FFT

(a) Suppose the input signal is $x_1(t) = \begin{cases} \sin(2\pi(100)t) & 0s \le t < 0.5s \\ \sin(2\pi(300)t) & 0.5s \le t < 0.75\ s \\ \sin(2\pi(100)t) & 0.75s \le t \le 2.0s \end{cases}$

Using Simulink and MATLAB, plot the FFT of $x_1(t)$ using block processing. Assume a sampling frequency of 1000 Hz and an FFT length of 256.

(b) Now change the input signal to $x_2(t) = \begin{cases} \sin(2\pi(100)t) & 0s \le t < 0.64s \\ \sin(2\pi(300)t) & 0.64s \le t < 0.89s \\ \sin(2\pi(100)t) & 0.89s \le t \le 2.0s \end{cases}$

Use the same sampling frequency and FFT length from part (a), plot the FFT of $x_2(t)$ using block processing, and comment on the results.

(c) Repeat part (b) but overlap the input data by 128 so that every block of 256 data points consists of 128 data points from the previous FFT calculation and 128 new data points.

(d) Repeat part (c) using an overlap of 192 so that every block of 256 data points consists of 192 data points from the previous FFT calculation and 64 new data points.

Solution

The Simulink model used for Example 8.3 (see Figure 8.3) is also used for this example.

(a) The MATLAB commands for part (a) are identical to those of Example 8.3(a) with the following exception:

```
x=sin(2*pi*100*t);
x(500:750)=sin(2*pi*300*t(500:750));
```

(b) Change the input signal to $x_2(t)$ using the following MATLAB commands, re-run the simulation, then plot the results:

```
x=sin(2*pi*100*t);
x(640:890)=sin(2*pi*300*t(640:890));
input=[t' x'];
```

Re- run FFT Block Simulation then run the following commands in MATLAB

```
Clear yfft; yfft(:,:)=yout(:,1,:);
mesh(F,K,abs(fftshift(yfft,1)))
```

(c) In the Simulink model, open up the Short-Time FFT block and change the overlap to 128. Re-run the simulation model and plot the results.

(d) In the Simulink model, open up the Short-Time FFT block and change the overlap to 192. Re-run the simulation model and plot the results.

The spectral plots for this example are shown in Figure 8.5 (a)-(d).

Comments on results of Example 8.4:

(a) As shown in Figure 8.5(a), the spectral plot for part (a) is a very nice representation of the spectrum of the signal $x_1(t)$. It clearly shows that the signal is initially a 100 Hz tone, switches to a 300 Hz tone, and then reverts back to a 100 Hz tone.

(b) The only difference between the signals $x_1(t)$ and $x_2(t)$ is the interval of time during which these signals become 300 Hz sine waves. The signal $x_1(t)$ is a 300 Hz sine wave from 0.5 – 0.75 sec while the signal $x_2(t)$ is a 300 Hz sine wave from 0.64 – 0.89 sec. Even though both signals have a 300 Hz component for exactly the same length of time, both signals were sampled at exactly the same rate, and the same length FFT was used in both cases, the spectrum for $x_2(t)$ shown in Figure 8.5(b) is quite different from the spectrum of $x_1(t)$. The spectrum in Figure 8.5(b) would indicate that $x_2(t)$ begins as a 100 Hz tone, then becomes a two tone signal (100 Hz and 300 Hz), and finally reverts back to a 100 Hz tone. What causes the big difference in results between parts (a) and (b)? To answer this question, consider the windows of time over which the FFT is being calculated and what the input signal samples look like within these windows of time. This information is shown in Table 8.1. In the case of $x_1(t)$, most of the samples of the 300 Hz section of the signal fall into the same buffer and just about fill it. In the case of $x_2(t)$, samples of the 300 Hz section get split between two adjacent buffers and fill only about half the buffer. The other half of the buffer is filled with samples of the 100 Hz section. Thus, the FFT calculation for these buffers indicates a two tone signal rather isolating the 300 Hz tone.

(c) In part (c), an overlap of 128 is introduced for the FFT processing. Each sampling window consists of 128 sample values from the previous window and 128 new sample values. This creates overlapping windows of time, but does not reduce the size of the FFT so the frequency resolution remains the same. As shown in Table 8.2 and also reflected in the spectral plot in Figure 8.5(c), this overlap allows the 300 Hz tone to be effectively isolated during one sampling window. The spectrum still indicates a two tone signal in the adjacent sampling windows.

(d) In part (d), the overlap is increased to 192 which means each input buffer consists of 192 past sample values and 64 new sample values. The resulting spectrum plotted in Figure 8.5(d) shows better time resolution as the signal transitions from 100 Hz to 300 Hz then back to 100 Hz.

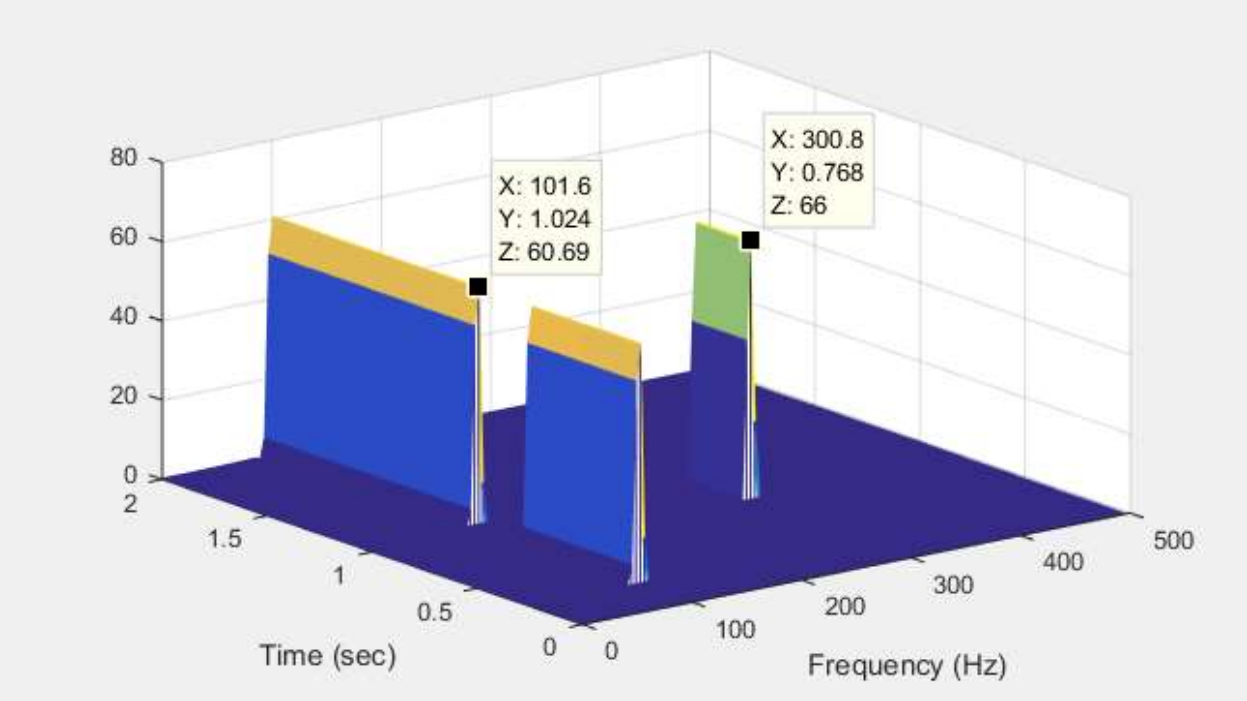


Figure 8.5a

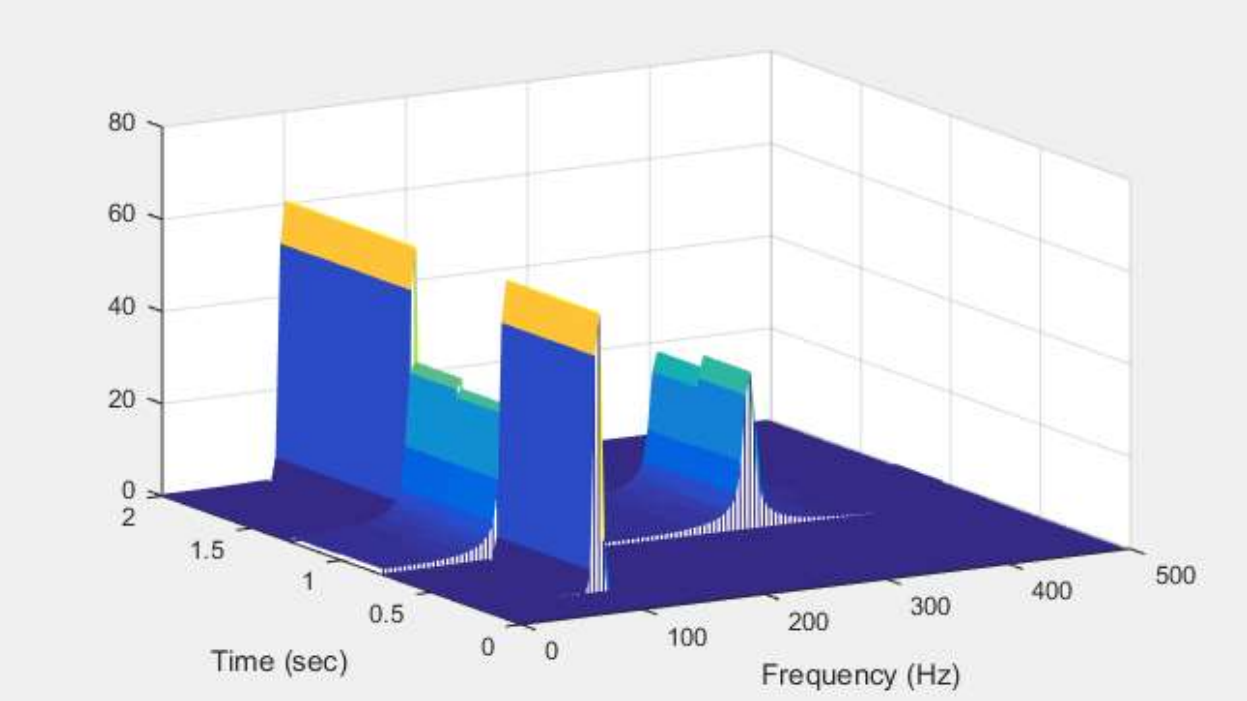


Figure 8.5b

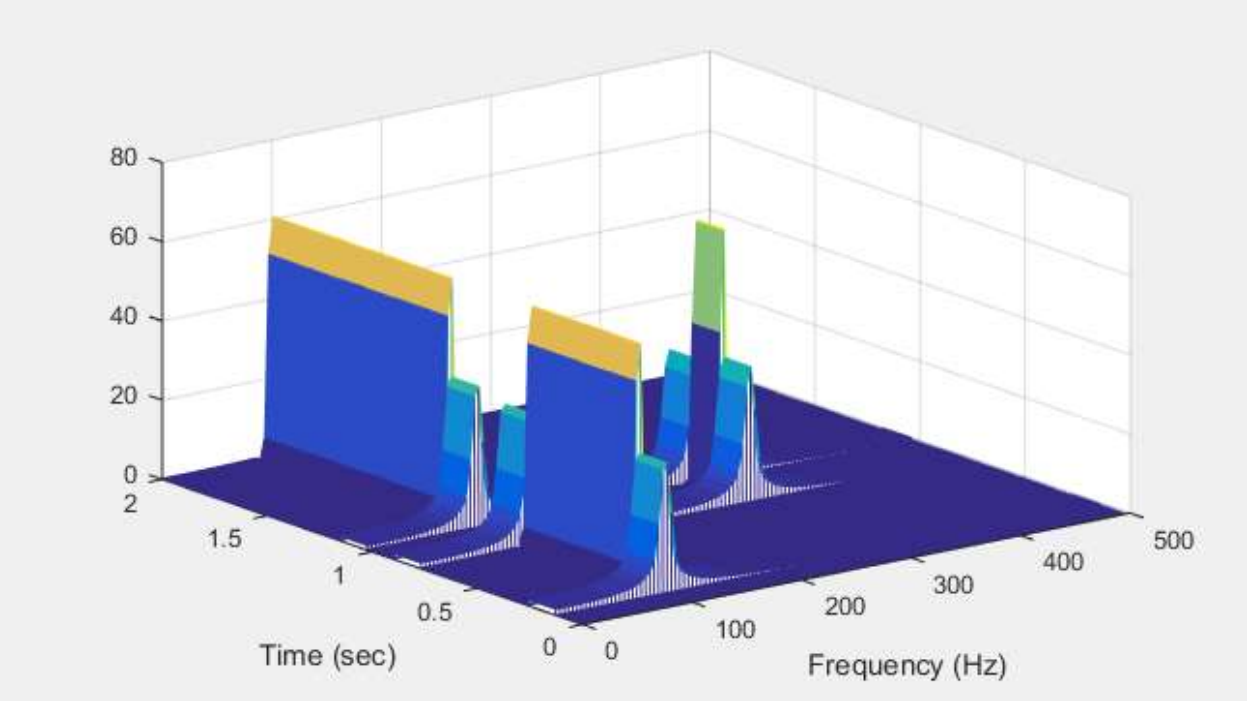


Figure 8.5c

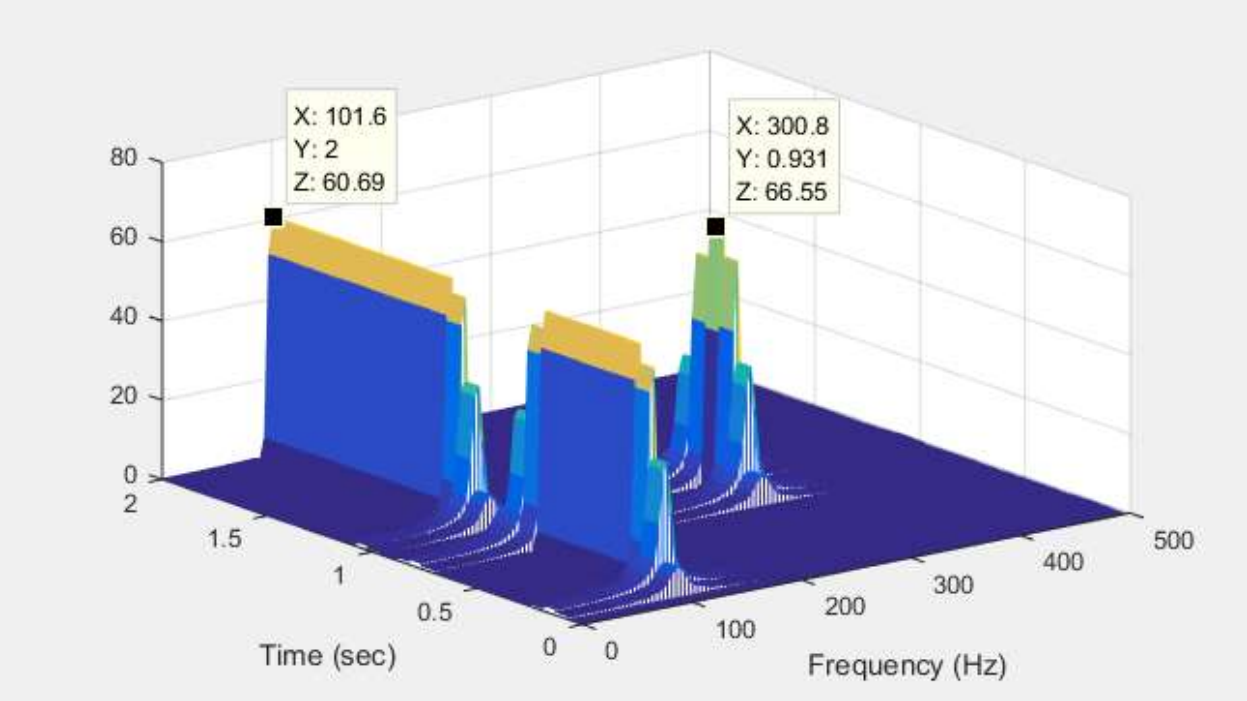


Figure 8.5d

Figure 8.5: Spectral Plots for Example 8.4

Table 8.1: Sample Intervals and Input Signal Content for Example 8.4(a)-(b)

Time Interval for FFT	$x_1(t)$		$x_2(t)$	
	Number of 100 Hz Samples	Number of 300 Hz Samples	Number of 100 Hz Samples	Number of 300 Hz Samples
0 – 0.256 s	256	0	256	0
0.256 – 0.512 s	244	12	256	0
0.512 – 0.768 s	18	238	128	128
0.768 – 1.024 s	256	0	134	122
1.024 – 1.28 s	256	0	256	0
1.28 – 1.536 s	256	0	256	0
1.536 – 1.8 s	256	0	256	0
1.8 – 2.046 s	256	0	256	0

Table 8.2: Sample Intervals and Input Signal Content for Example 8.4 (c)

Time Interval for FFT	$x_2(t)$	
	Number of 100 Hz Samples	Number of 300 Hz Samples
0 – 0.256 s	256	0
0.128 – 0.384 s	256	0
0.256 – 0.512 s	256	0
0.384 – 0.64 s	256	0
0.512 – 0.768 s	128	128
0.640 – 0.896 s	6	250
0.768 – 1.024 s	134	122
0.896 – 1.152 s	256	0
1.024 – 1.28 s	256	0
Remaining Intervals	256	0

Challenge Question 8.2
Example 8.4 shows that increasing the overlap of input sample values for spectral analysis results in a better reflection of how the signal spectrum varies over time. What is the disadvantage of increasing the overlap?

8.4 SLIDING DISCRETE FOURIER TRANSFORM

The sliding DFT is a real-time implementation of a DFT. The sliding DFT computes an N-point spectrum at each sampling instant; that is, every time a new data sample is taken. This would be equivalent to the FFT block processing in the previous section using an overlap of N – 1 for an N-point spectrum. The advantages of overlap were illustrated in Example 8.4; however, computing an N-point FFT every single sampling instant is much too complex. The sliding DFT is a practical algorithm for computing the spectrum at each sampling instant. The DFT was introduced in Chapter 7 and is computed as follows:

N-PT DFT

At $t = LT_s$ (i.e., end of a sampling interval), let x(k) be the N input samples taken over the last sampling interval; that is, $(L-1)T_s < t \leq LT_s$.

$$x(k) = [\, x(L-(N-1)T_s) \ldots x\big((L-1)T_s\big)\ x(LT_s)\,]$$

$$X_{DFT}(nf_o) = \sum_{k=0}^{N-1} x(k)e^{-j2\pi nk/N} \qquad n = 0, 1, \ldots N-1$$

Bin Frequencies: $0, f_o, 2f_o, \ldots (N-1)f_o$ where $f_o = F_s/N$

(8.3)

Computing a DFT for each frequency bin simply involves multiplying N consecutive input data values by the exponential weighting functions for that bin then adding the results. It is really easy to understand how the sliding DFT algorithm was derived using the graphical illustration of an 8-pt DFT shown in Figure 8.6. The discrete frequencies (or bin frequencies) for an 8-pt DFT are 0, f_o, $2f_o$, $3f_o$, $4f_o$, $5f_o$, $6f_o$, $7f_o$. The exponential weights for Bin 0 (f = 0 Hz) are all one so that case is not drawn. The exponential weights for Bin 1 ($f = f_o$ Hz) are the eight weights equally distributed clockwise about the unit circle as shown in Figure 8.6. The exponential weights for Bin 2 ($f = 2f_o$ Hz) are the eight weights obtained by taking every other weight in Bin 1 starting with one, moving clockwise, and wrapping around the circle as many times as necessary. The exponential weights for Bin 3 ($f = 3f_o$ Hz) are the eight weights obtained by taking every 3rd weight in Bin 1 starting with one, moving clockwise, and wrapping around the circle as many times as necessary. This same pattern would continue to find the weights for Bins 4-7.

Given eight consecutive data samples, the DFT for each bin frequency is computed by multiplying the eight data values by the weights for that bin and adding the results. In Figure 8.6, it is assumed that the eight consecutive data samples are numbered 0, 1, 2, … 7. These sample numbers are shown in the figure next to their corresponding weights.

Challenge Question 8.3

For a 16-pt DFT, assume the sixteen consecutive data samples are numbered 0, 1, 2, … 15. Sketch a diagram similar to the one in Figure 8.6 that shows how the data samples are distributed among the sixteen weights for Bin 5.

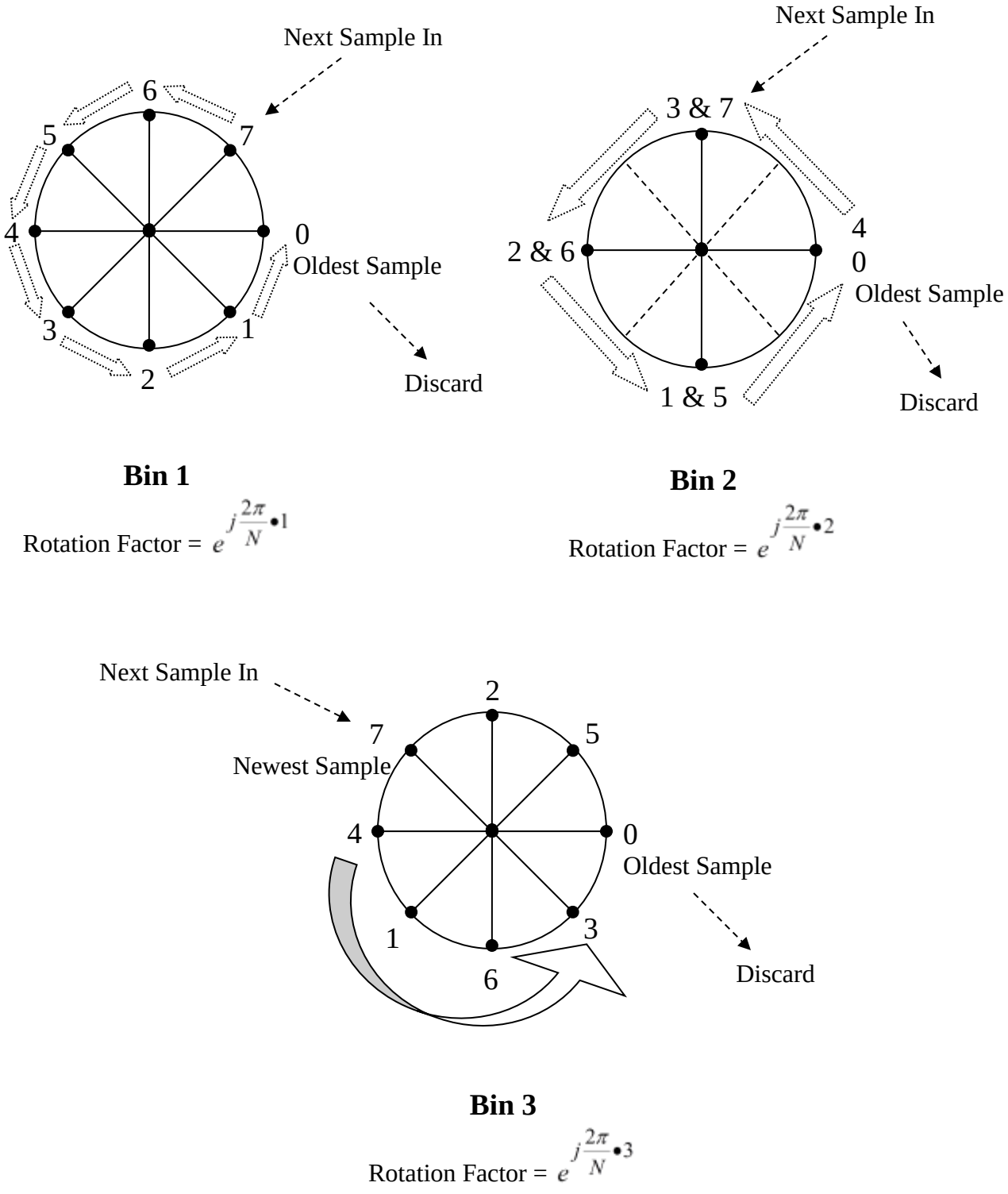


Figure 8.6: Graphical View of Sliding DFT

Assume the DFT has been calculated for all eight bin frequencies using the data samples 0, 1, 2, … , 7. Now a new data sample value is taken and we wish to compute a new DFT using the eight most recent sample values; that is, 1, 2, 3, … 8. While it would certainly be possible to start all over again and compute a new DFT, this would not be smart. Looking at the diagrams in Figure 8.6, three things must happen to find the new DFT:

1. Discard the oldest data sample
2. Rotate the remaining samples to their new corresponding weights
3. Account for the newest data sample

Since the weight on the oldest sample is always one, the oldest sample can be discarded by simply subtracting it directly from the previous DFT calculation. The new weights for the remaining samples are derived by simply shifting each data sample counterclockwise to the next weight in the bin which would be a $\frac{2\pi}{N}n$ radian shift (*n* is the bin number). Since all the data samples for a specified bin frequency are shifted by the same amount, the product of each sample times its new weight equals the old product times the rotation factor, $e^{j\frac{2\pi}{N}n}$, for Bin *n*.

So, at this point, we have

$$\text{Bin } n: \quad \text{NEXT DFT} = e^{j\frac{2\pi}{N}n}[\text{PREVIOUS DFT} - \text{oldest data sample}] + ?$$

The only thing not accounted for at this point is inserting the newest data sample multiplied by its corresponding weight. Looking at Figure 8.6, the weight corresponding to the newest data sample just happens to be equivalent to the rotation factor. So, we now have the following:

$$\text{NEXT DFT} = e^{j\frac{2\pi}{N}n}[\text{PREVIOUS DFT} - \text{oldest data sample}] + e^{j\frac{2\pi}{N}n}[\text{newest data sample}]$$

which simplifies to

$$\text{NEXT DFT} = e^{j\frac{2\pi}{N}n}[\text{PREVIOUS DFT} - \text{oldest data sample} + \text{newest data sample}]$$

In more conventional math notation, the algorithm for a sliding DFT is

SLIDING DFT

$$X_{nf_o}(L) = e^{j2\pi n/N}\left[X_{nf_o}(L-1) - x(L-N) + x(L)\right] \qquad n = 0, 1, \ldots N-1 \tag{8.4}$$

The algorithm must be initialized in order to implement the sliding DFT. There are two options for initializing the algorithm. One option is to collect the first N input data values then calculate an N-PT DFT that would be used as the initializing DFT in the difference equation above. Another option is to simply start at t = 0 by loading the input data buffer with all zeros (except the current input x(0)) and initializing $X_{nf_O}(L-1)$ to zero. In this case, it is important to recognize that the difference equation would not produce valid DFT information until the N^{th} iteration through the difference equation. Both options require the exact same time delay of NT_s before the first valid DFT can be computed. The second option is a lot easier to program because it simply requires implementation of the difference equation and does not include the additional computational burden of the one-time N-PT DFT calculation.

A block diagram realization of the sliding DFT is shown in Figure 8.7.

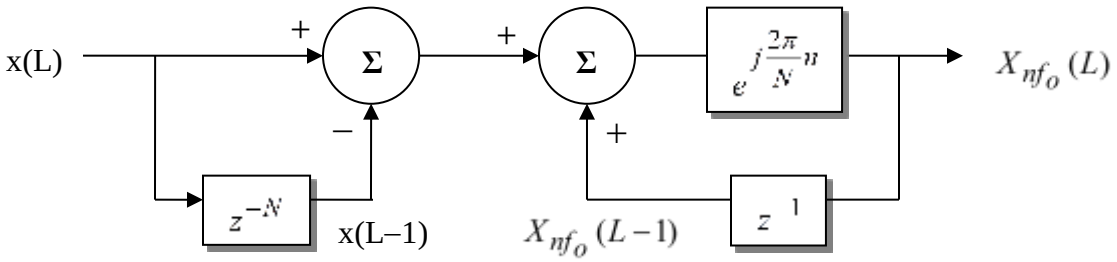


Figure 8.7: Block Diagram Realization of Sliding DFT

The first part of the block diagram in Figure 8.7 is simply a comb filter. The second part of the block diagram involves feedback of the previous output. The transfer function for the Sliding DFT is given by:

$$H(z) = \frac{[1-z^{-N}]\times e^{j\frac{2\pi}{N}n}}{[1-e^{j\frac{2\pi}{N}n}z^{-1}]} = \frac{[z^{N}-1]\times e^{j\frac{2\pi}{N}n}}{z^{N-1}[z-e^{j\frac{2\pi}{N}n}]} \qquad (8.5)$$

Challenge Question 8.4
What are the poles associated with the Sliding DFT algorithm? What does this indicate about the stability of the algorithm?

Modified Sliding DFT
As indicated in the challenge question, the Sliding DFT algorithm is marginally stable and may become unstable due to rounding of the coefficient, exp(j2πn/N). If stability is a problem, the sliding DFT algorithm can be modified to guarantee stability. The modified algorithm for a sliding DFT is given by:

Modified Sliding DFT (to guarantee stability)

$$X_{nf_o}(L) = r \cdot e^{j\frac{2\pi}{N}n}\left[X_{nf_o}(L-1) - r^N x(L-N) + x(L)\right] \qquad n = 0, 1, \ldots N-1$$

$$0 < r < 1$$

(8.6)

Notice that in the modified algorithm, the numerical coefficient which posed a potential stability problem has been scaled by a real number r < 1. This will guarantee stability of the algorithm as long as the magnitude of $re^{j\frac{2\pi}{N}n} < 1$. How does the scale factor, r, affect the DFT calculation? The modified sliding DFT algorithm will actually calculate the following:

$$X_{DFT}(nf_o) = \sum_{k=0}^{N-1} x(k) r^{N-k} e^{-j2\pi nk/N} \qquad n = 0 \ldots N-1 \tag{8.7}$$

Notice that the only difference between this modified DFT and the original DFT is the powers of r that have been inserted into the summation. These powers of r could be interpreted either as scaling factors that pull the bin weights inside the unit circle or as a non-symmetric unusual type of window on the input data samples as shown in Figure 8.8 for an 8-pt DFT. Notice that choosing r very close to one minimizes the scaling effect on the input data samples; that is, $r^{N-k} \approx 1$ so long as r is very close to 1.

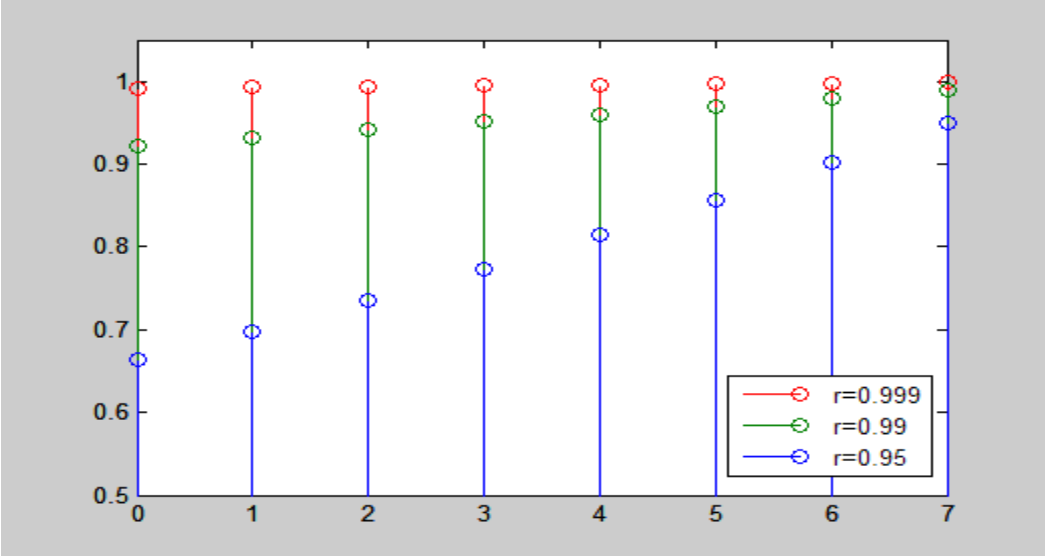


Figure 8.8: Effect of Scale Factor, r for 8-pt DFT

In summary, the scale factor r should be chosen as close to 1 as possible while still ensuring that the magnitude of $re^{j(2\pi/N)n}$ is less than 1 when implemented on the DSP processor. This will guarantee stability of the algorithm while minimizing the effect of scaling on the accuracy of the DFT results. A block diagram for the modified sliding DFT is shown in Figure 8.9.

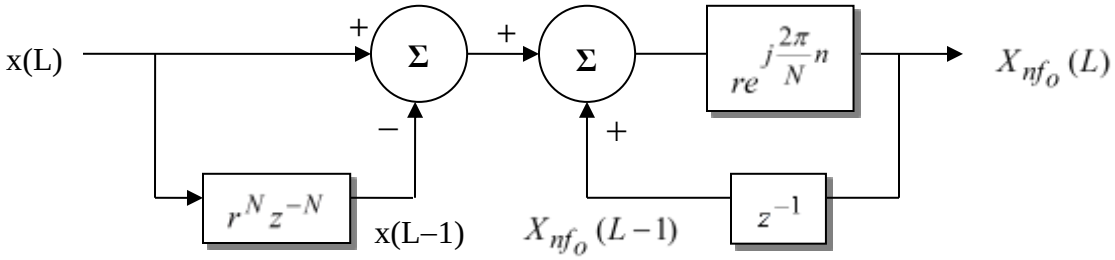


Figure 8.9: Block Diagram Realization of Modified Sliding DFT

Example 8.5: Tone Detection using Sliding DFT

The input signal for this sliding DFT example was generated by playing and recording a series of notes on a digital piano (see Figure 8.10(a)). The Midi recording was then converted to a wav file which was imported into MATLAB using the ***audioread*** function. A wav file uses a sampling rate of 44,100 Hz which is higher than necessary for this example so the data was decimated by a factor of 10 to reduce the sampling rate to 4410 Hz. A plot of the decimated wav file is shown in Figure 8.10(b). An FFT of the decimated wav file was computed using MATLAB and is plotted in Figure 8.10(c). The frequencies for the notes played are: Middle C = 261.63 Hz, E = 329.63 Hz, G = 392 Hz, and C one octave above Middle C = 523.25 Hz. Notice that the FFT in Figure 8.10(c) shows resonant peaks at the discrete frequency bins closest to these note frequencies.

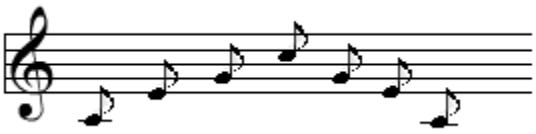


Figure 8.10a: Series of Notes Played (Middle C, E, G, C, G, E, Middle C)

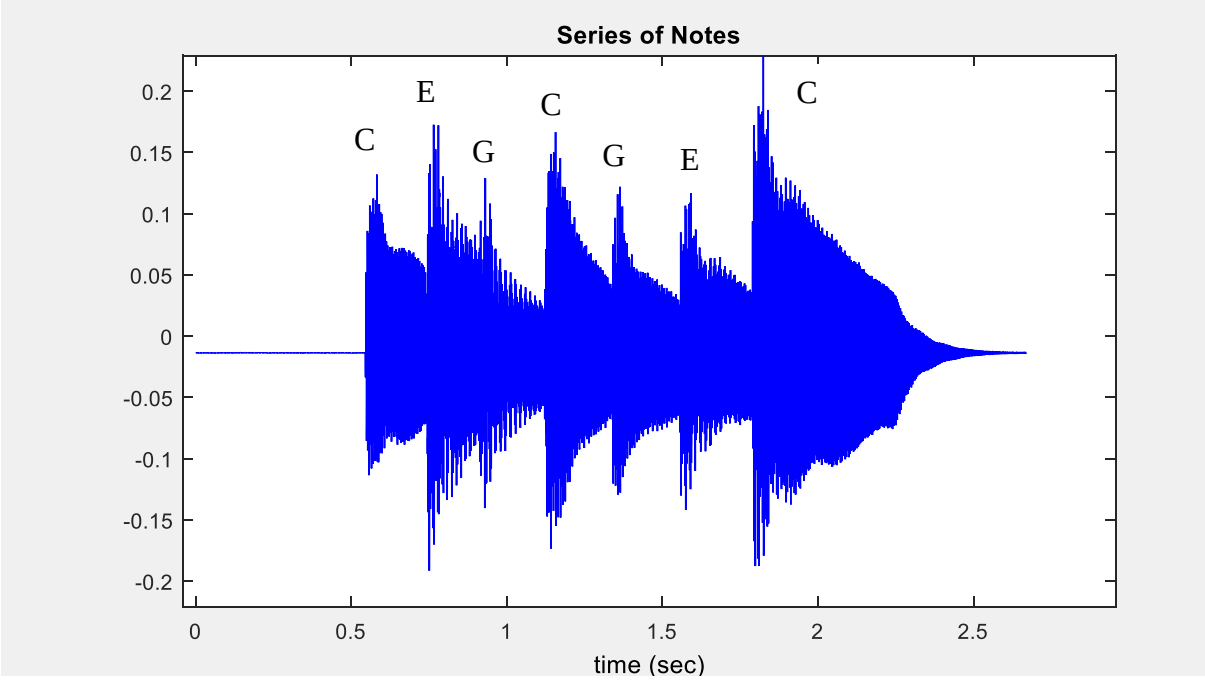


Figure 8.10b: Plot of wav file Data

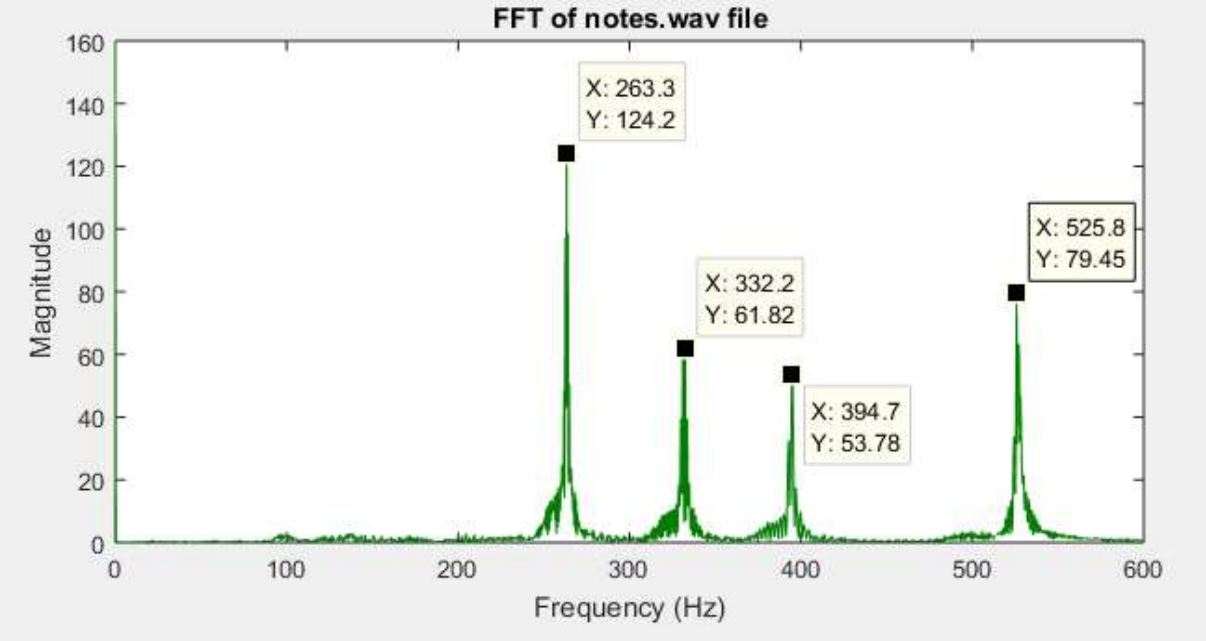


Figure 8.10c: FFT of the Decimated wav file

Figure 8.10: Example 8.5

The MATLAB commands used to generate the plots in Figure 8.10b and 8.10c are listed in Figure 8.11.

```
% Example 8.5
[y,Fs] = audioread('notes.wav');
% Decimate the signal by 10 giving a new sampling rate of 4410
% Just include first channel of stereo signal
ydec=decimate(y(:,1),10);
tdec = 0:1/4410:(size(ydec)-1)*1/4410;
plot(tdec,ydec);

fs=Fs/10;
yf=fftshift(fft(ydec));
N=length(yf); % N= # of time samples = # of freq. comp in fft
fo=fs/N;      % Calculate resolution of fft in Hz
f=-floor(N/2)*fo:fo:(ceil(N/2)-1)*fo; % Set up DFT Frequencies

%Plot the magnitude response of fft
figure;
plot(f,abs(yf));
title('FFT of notes.wav file')
xlabel('Frequency (Hz)');xlim([0 600])
```

Figure 8.11: MATLAB Code for Figure 8.10 (b) and (c)

Although the spectral plot in Figure 8.10c does indeed accurately reflect the notes that were played, it gives no indication of when each note was played or the duration of each note. For the next part of this example, a 256 pt-sliding DFT will be implemented to show how the signal spectrum varies over time. The resolution for the 256-pt DFT is

$$f_o = \frac{F_s}{N} = \frac{4410}{256} = 17.2265625 Hz$$

The sliding DFT could easily be calculated for all 256 bin frequencies but in this example it will only be calculated for the bin frequencies closest to the notes played. The bin numbers for each note are shown in Table 8.3.

Table 8.3: Bin Numbers for Notes Played

NOTE	FREQUENCY	FREQUENCY/ f_o	BIN #
Middle C	261.63	15.2	15
E	329.63	19.1	19
G	392	22.76	23
C octave above Middle C	523.25	30.4	30

The results are plotted in Figure 8.12. Each of the sliding DFTs correlates very well with the original wav file. It is possible to examine each of the sliding DFTs in Figure 8.12 and determine when each of the individual notes was played. The bottom graph for the C one octave above Middle C does require comment. It appears that this note was played in conjunction with Middle C at t = 0.6 sec and t = 1.8 sec. This is actually due to how the digital piano works. The digital piano provides several options for special effects; one of which is that when a note is played, harmonics are also produced. The C above Middle C is the 2nd harmonic of Middle C and will be generated every time Middle C is played (with smaller amplitude).

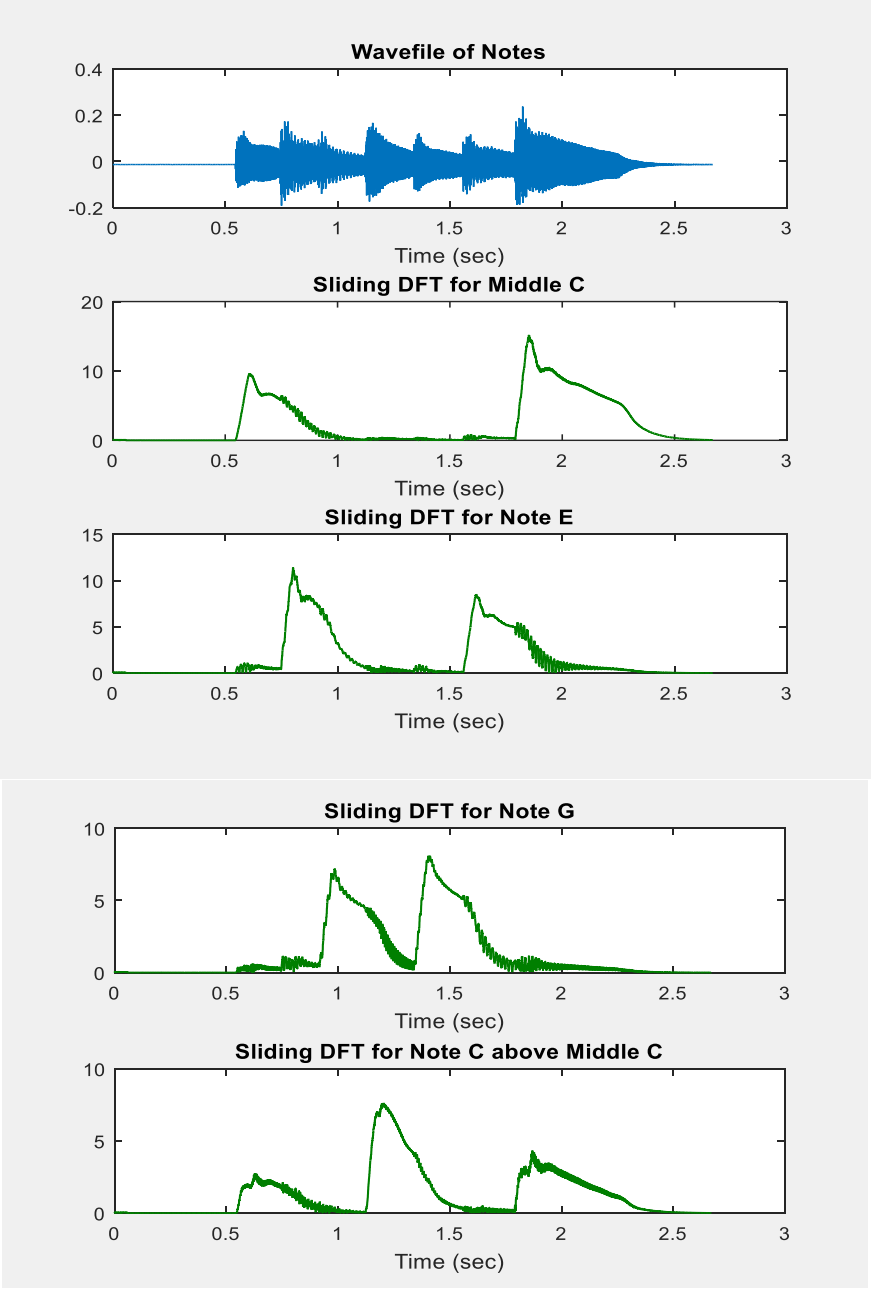


Figure 8.12: Sliding DFT Results (Example 8.5)

.

```
[y,Fs] = audioread('notes.wav');
ydec=decimate(y(:,1),10); % Decimate Channel 1 of input wav file by 10
tdec = 0:1/4410:(size(ydec)-1)*1/4410;
subplot(3,1,1);plot(tdec,ydec);
title('Wavefile of Notes');xlabel('Time (sec)')

% Sliding DFT
N = 256;  % Size of DFT
x = [zeros(1,N) ydec' ]; % Initialize input data vector with N zeros

% Middle C = Bin 15
n = 15; X_15(1)=0; % Set Bin # and Initialize first DFT to zero
for m = 1:length(x)-N-1,
    X_15(m+1)=exp((1i*2*pi/N)*n)*[X_15(m)-x(m)+x(m+256)];
end
subplot(3,1,2); plot(tdec,abs(X_15));
title('Sliding DFT for Middle C');xlabel('Time (sec)')

% Note E above Middle C = Bin 19
n = 19; X_19(1)=0;
for m = 1:length(x)-N-1,
    X_19(m+1)=exp((1i*2*pi/N)*n)*[X_19(m)-x(m)+x(m+256)];
end
subplot(3,1,3); plot(tdec,abs(X_19));
title('Sliding DFT for Note E');xlabel('Time (sec)')

% Note G above Middle C = Bin 23
n = 23; X_23(1)=0;
for m = 1:length(x)-N-1,
    X_23(m+1)=exp((1i*2*pi/N)*n)*[X_23(m)-x(m)+x(m+256)];
end
figure; subplot(2,1,1); plot(tdec,abs(X_23));
title('Sliding DFT for Note G');xlabel('Time (sec)')

% C above Middle C = Bin 30
n=30; X_30(1)=0;
for m = 1:length(x)-N-1,
    X_30(m+1)=exp((1i*2*pi/N)*n)*[X_30(m)-x(m)+x(m+256)];
end
subplot(2,1,2); plot(tdec,abs(X_30));
title('Sliding DFT for Note C above Middle C');xlabel('Time (sec)')

% Show all 4 sliding DFTs in a single plot
figure; subplot(2,1,1); plot(tdec,ydec);
title('Input Data: Wavefile of Notes');xlabel('Time (sec)')
subplot(2,1,2);
plot(tdec,abs(X_15),tdec,abs(X_19),tdec,abs(X_23),tdec,abs(X_30));
title('Sliding DFTs for all Four Bins');xlabel('Time (sec)')
legend('Middle C','E','G','C octave above Middle C')
```

Figure 8.13: MATLAB Code for Sliding DFT (Example 8.5)

Figure 8.14 shows all four sliding DFTs in a single plot.

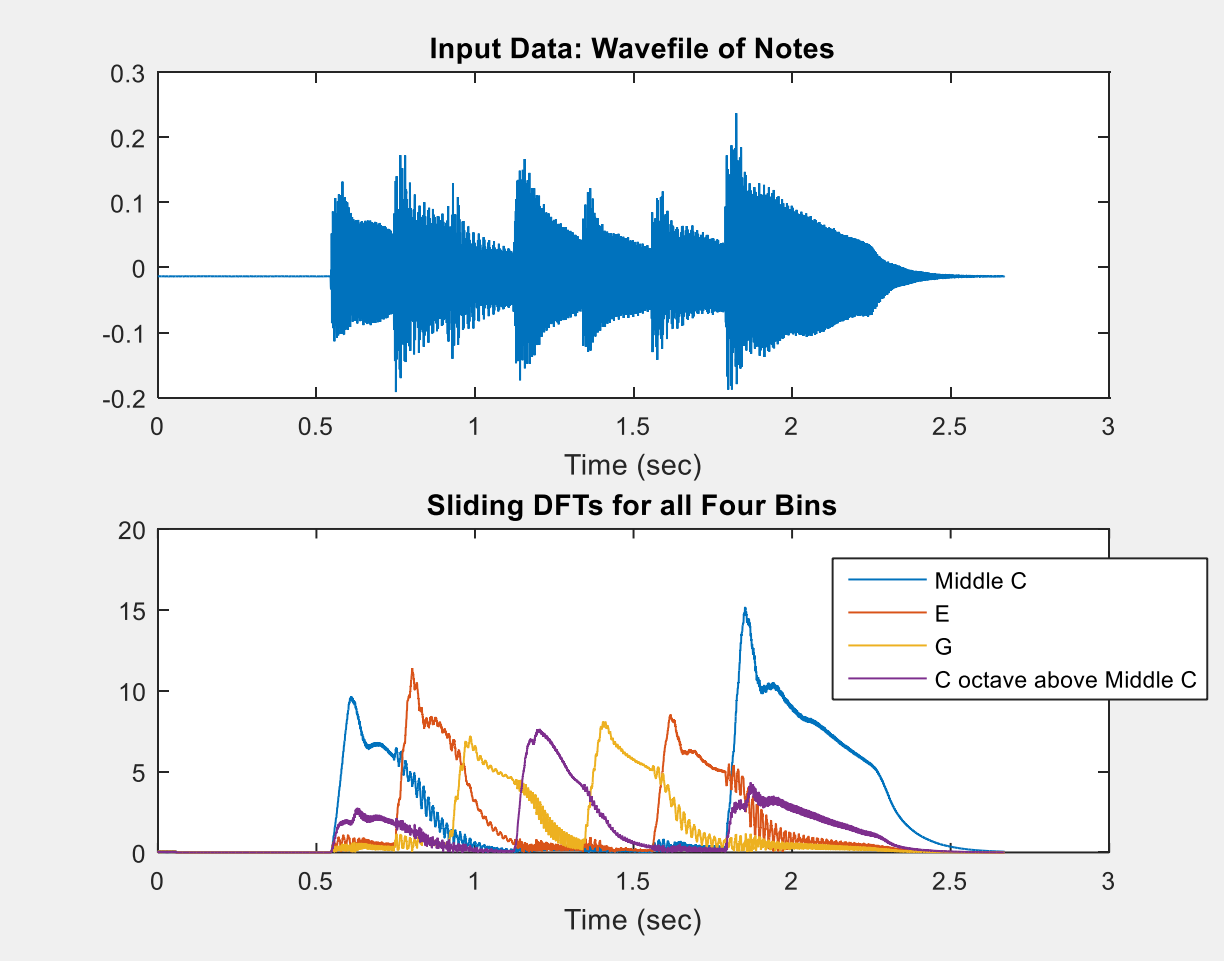


Figure 8.14: Combined Sliding DFT Results (Example 8.5)

Effect of Time Duration, D

In the previous section on real-time spectral analysis using block processing and FFTs, it was mentioned that the choice of time duration, D, was critical. Choosing a time duration too long could result in missing short burst changes in the input signal (poor time resolution), but decreasing the time duration reduces the frequency resolution. This issue was illustrated in Example 8.3. The same issue occurs with a sliding DFT. Even though the spectrum is updated at every sampling instant, the spectrum is still based on N samples of the input signal spanning a time duration of $D = N \cdot F_s$.

Wavelets are an alternative to FFTs or sliding DFTs for real-time spectral analysis. Wavelets attempt to bridge the tradeoff between poor time resolution with large D and poor frequency resolution with small D by offering good time resolution at higher frequencies and good frequency resolution at lower frequencies. The underlying assumption on the input signal is that the lower frequencies persist for longer intervals of time (so D can be longer); whereas, the higher frequencies tend to occur in shorts bursts (so D must be shorter).

Wavelet processing essentially breaks the input signal into distinct frequency bands through a series of digital filters and provides good frequency resolution in the lower bands and good time resolution in the higher bands of the signal. A detailed analysis of wavelet processing is beyond the scope of this text, but it is a fascinating subject and there are many good texts and websites available to the interested reader.

Sliding DFT with Windowing

In section 7.2, the importance of windowing blocks of data to enhance spectral analysis by reducing the effect of leakage was discussed. With the Sliding DFT, windowing the incoming data samples directly (in the time domain) is not an option.

Challenge Question 8.5
Why is it not possible to window the incoming data samples directly for a sliding DFT?

Although it is not possible to window the incoming input data samples in the time domain, it is possible to produce the same effect by performing a convolution in the frequency domain. Recall the following useful property first introduced in Chapter 4:

$$x(t)w(t) \leftrightarrow X(f) * W(f) \tag{8.8}$$

Multiplication of two signals in the time domain is equivalent to convolving the spectrums of the two signals. One of the common window functions introduced in Chapters 5 and 7 is the Hamming window described by

$$w(kT_s) = 0.54 - 0.46\cos\left(\frac{2\pi kT_s}{N-1}\right) \quad k = 0 \ldots N-1 \tag{8.9}$$

This window function has a very simple Fourier Transform given by:

$$W(f) = 0.54\delta(f) - 0.23\left[\delta(f + f_o) + \delta(f - f_o)\right] \qquad \text{where} \quad f_o = \frac{F_s}{N} \tag{8.10}$$

Assuming that the sliding DFT provides an accurate sampled approximation to the spectrum of the input data signal X(f) at the bin frequencies nf_o, and applying the convolution property using W(f) for the Hamming window yields

$$X^W(nf_o) = 0.54X(nf_o) - 0.23\left[X((n+1)f_o) + X((n-1)f_o)\right] \tag{8.11}$$

In other words,

$$\text{Bin } n \text{ Windowed} = 0.54\,[\text{ Bin } n] - 0.23\,[\text{ Bin } (n-1) + \text{Bin } (n+1)\,] \tag{8.12}$$

Windowing is therefore accomplished by using the sliding DFT algorithm to compute the spectrum at the various bin frequencies, then combining adjacent bins to produce a spectrum for windowed data samples.

Several other window functions were explored in Chapters 5 and 7. Any window function that can be represented as a combination of cosine or sine functions could be used in place of the Hamming window. Table 8.4 provides some common window functions and the corresponding "windowed" DFT.

Table 8.4: DFT using Common Window Functions

WINDOW	WINDOW FUNCTION	WINDOWED DFT
Hamming	$w(k) = 0.54 - 0.46\cos\left(\frac{2\pi k}{N-1}\right)$	$X_n^w = 0.54X_n - 0.23\left[X_{(n+1)} + X_{(n-1)}\right]$
Hanning	$w(k) = 0.5 - 0.5\cos\left(\frac{2\pi k}{N-1}\right)$	$X_n^w = 0.5X_n - 0.25\left[X_{(n+1)} + X_{(n-1)}\right]$
Blackman	$w(k) = 0.42 - 0.5\cos\left(\frac{2\pi k}{N-1}\right) + 0.08\cos\left(\frac{4\pi k}{N-1}\right)$	$X_n^w = 0.42X_n - 0.25\left[X_{(n+1)} + X_{(n-1)}\right] + 0.04\left[X_{(n+2)} + X_{(n-2)}\right]$

Notice that the coefficients for the Hanning window can be used to advantage when implementing this window. Multiplication by 0.5 can be accomplished by a single bit shift to the right and multiplication by 0.25 can be implemented by a 2 bit shift to the right.

8.5 ADAPTIVE FILTERING

Adaptive filtering is the process of changing the weights of a filter in real-time based on some type of input/output signal information. Echo cancellation, room acoustic identification, and noise cancellation are examples of adaptive filtering applications.

Echo Cancellation

Echo cancellation is extremely useful in telecommunications. The two types of echo are hybrid echo and acoustic echo. Hybrid echo occurs in a wire-line telephone network switch where the two-wire analog loop connection to the house is switched to a four-wire digital line for transmission. The echo is simply a result of an impedance mismatch in the two-wire to four-wire conversion circuitry. Hybrid echo occurs when one or more speakers are utilizing a land line connection. Acoustic echo results from unwanted feedback between a speaker and a microphone or from reflection of audio signals from surfaces back to the microphone. Acoustic echo occurs in very small phones where the earpiece and microphone are not sufficiently isolated or in "hands-free" conversations. Digital signal processors programmed as adaptive filters are used to cancel unwanted echo.

A block diagram for Hybrid echo cancellation is shown in Figure 8.15. When Speaker #1 talks, the impedance mismatch at the hybrid causes an echo of his or her speech. In the absence of echo cancellation, this echo would be combined with a voice signal (if present) from Speaker #2 and would be transmitted back to Speaker #1. The adaptive FIR filter in the hybrid produces an estimate of the echo and subtracts the estimate from the return signal effectively cancelling the echo. The weights of the adaptive FIR filter are adjusted in real-time based on the incoming signal from Speaker #1 and the residual error for the echo estimate. The filter weights are only adjusted when Speaker #1 is speaking and Speaker #2 is not in which case a residual error of zero would indicate a perfect estimate of the hybrid echo signal. Typically a least mean squares (LMS) algorithm (or version of such) is used to adjust the filter weights to minimize the residual error. The LMS algorithm is covered later in this section.

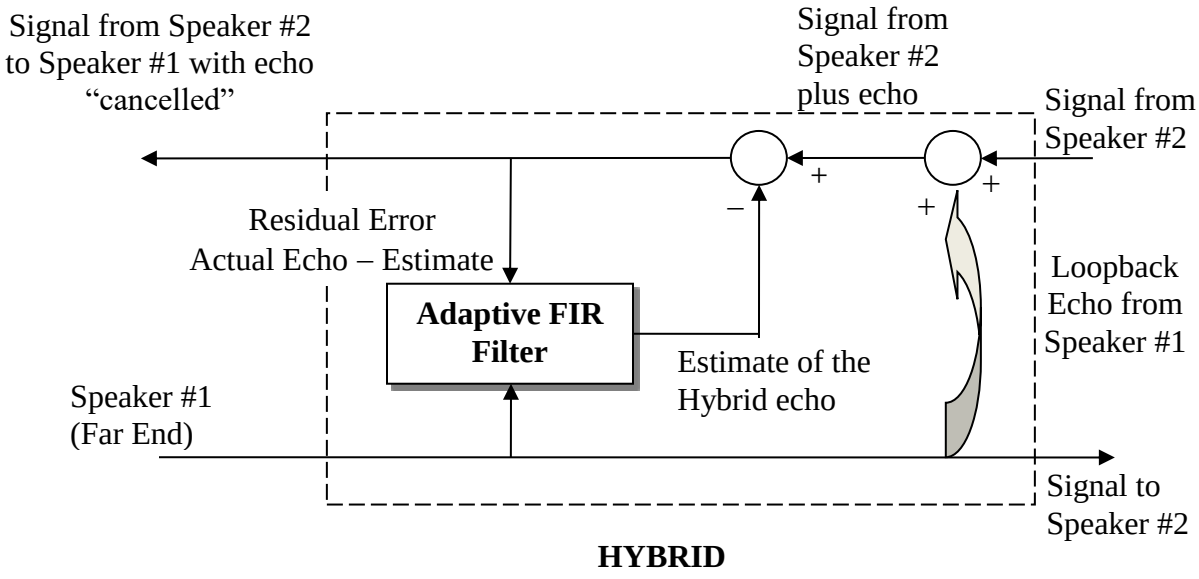


Figure 8.15: Echo Cancellation

The hybrid echo signal is relatively easy to cancel because it is linear and stationary; that is, the echo signal is simply the original speech signal attenuated and time-delayed. Acoustic echo is much more difficult to cancel because it is non-linear and non-stationary. Methods for cancelling acoustic echo vary from vendor to vendor with varying degrees of success (or failure).

Room Acoustic Identification

Adaptive filters can also be used to identify the acoustics of an enclosed space such as a room, an auditorium, a concert hall or even a stairwell. Acoustic identification is useful for new construction, for modifying existing spaces, and for reverb filter design. A basic block diagram

for room acoustic identification is shown in Figure 8.16. An input signal, typically white or pink noise, is applied to an adaptive filter and to a loudspeaker in the room. The signal, modified by the acoustical characteristics of the room, is picked up by the microphone. This signal is compared to the output of the adaptive filter producing a difference or error signal. The filter weights are adjusted based on the applied input signal and on the error signal in order to minimize the error signal. If the error signal converges to zero, then the adaptive filter weights "match" the impulse of the room.

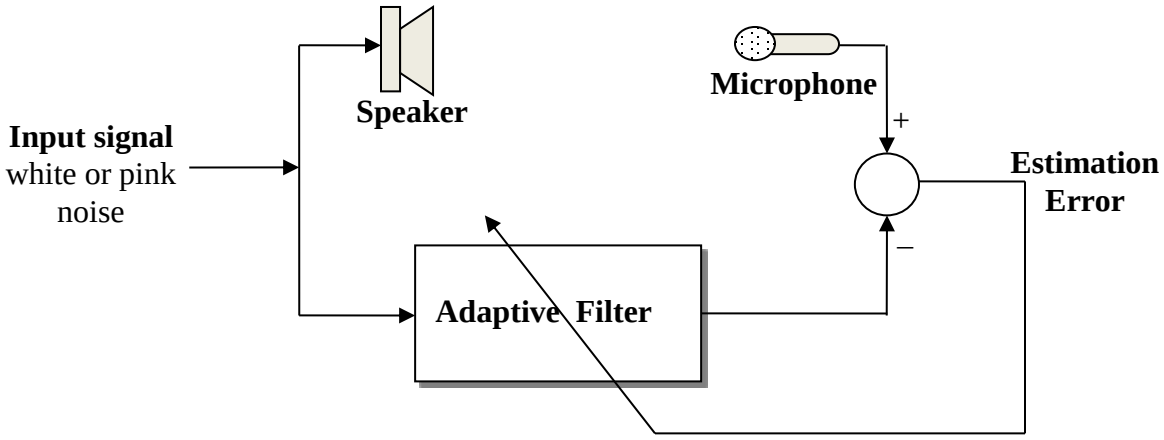


Figure 8.16: System Identification

Noise Cancellation

Adaptive filters can be used to suppress or cancel noise in a signal or in the surrounding environment. Noise-cancelling headphones reduce the level of unwanted ambient noise from fans or engines or other background noise sources. There are numerous biomedical applications for adaptive filtering. One of the earliest biomedical applications for noise cancellation was by Widrow et. al to eliminate the 50 Hz "hum" from an ECG (electrocardiograph) heart signal. Widrow also utilized adaptive filtering to eliminate the mother's heartbeat signal from a fetal monitor signal in order to produce the heartbeat of the fetus only.

A block diagram for adaptive noise cancellation is shown in Figure 8.17. The reference noise signal is applied to an adaptive filter to produce an "anti-noise" signal that is then subtracted from the noisy signal to reduce the noise level in the signal. In the case of noise-cancelling headphones, the reference noise signal is the background noise picked up by a microphone. For the fetal heartbeat application, the reference noise signal is the mother's heartbeat signal picked up by a second monitor.

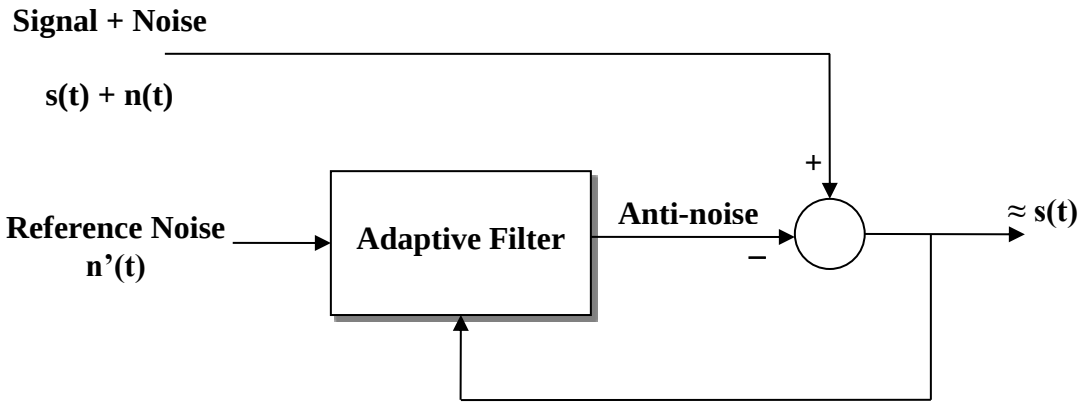


Figure 8.17: Noise Cancellation

Least Mean Squares Algorithm

One algorithm for adjusting the weights of an adaptive filter is the least mean squares (LMS) algorithm. The block diagram in Figure 8.18 is used to define the symbols used in the algorithm for the various signals.

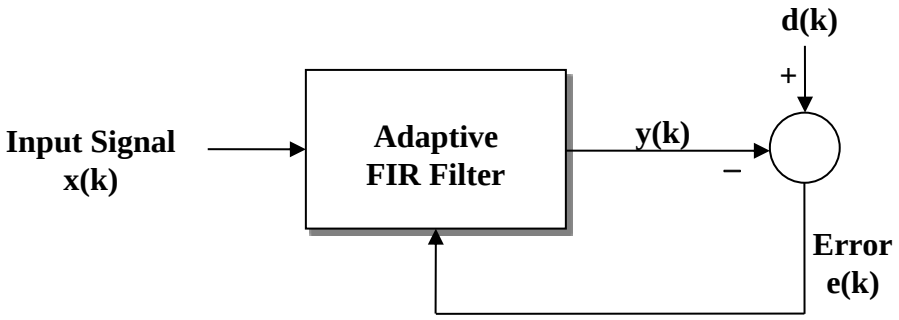


Figure 8.18: Adaptive FIR Filter

The weights (or filter coefficients) of the adaptive FIR filter are adjusted to minimize the error, e(k), according to some constraint. The error signal is simply the difference between the output of the adaptive filter and the desired output. Using w_0, w_1, … w_{N-1} to designate the filter weights, the output of the adaptive filter is

$$y(k) = w_0(k)x(k) + w_1(k)x(k-1) + \ \ldots\ w_{N-1}(k)x\big(k-(N-1)\big).$$

The LMS algorithm minimizes the expected value of the square of the error signal; that is the power of the error signal, and results in the following update for the filter coefficients:

LMS Algorithm

$$w_i(k+1) = w_i(k) + \mu e(k) \cdot x(k-i) \quad for \;\; i = 0 \;\ldots\; N-1$$

e(k) = d(k) – y(k) = estimation error
μ: adaptation step size
N: order of FIR filter
$w_i(k)$: i^{th} filter coefficient at time kT_s (8.13)

The adaptation step size, μ, affects how much the filter weights will be changed at each iteration of the algorithm and therefore affects the rate of convergence of the algorithm. Choosing a larger value for μ could result in faster convergence (adaptivity) but could also cause less accurate filter weights, larger estimation error, and affect the stability of the algorithm. The adaptation step size is sensitive to the characteristics of the input signal which means there is no "one size fits all" value for μ. The normalized LMS algorithm reduces input signal sensitivity by normalizing the weight adjustment term by the input signal power as shown in Equation 8.14:

Normalized LMS Algorithm

$$w_i(k+1) = w_i(k) + \mu e(k) \cdot \frac{x(k-i)}{\sum_{i=0}^{N-1} x^2(k-i)} \qquad for \;\; i = 0 \;\ldots\; N-1 \qquad (8.14)$$

Recall in the previous section, it was noted that the Sliding DFT algorithm was marginally stable (poles on unit circle) and a scale factor was introduced to stabilize the algorithm. The LMS algorithms are also marginally stable with a discrete pole at 1. A leakage factor can be added to stabilize the algorithm:

Normalized LMS Algorithm with Leakage Factor

$$w_i(k+1) = (1-\mu\alpha) w_i(k) + \mu e(k) \cdot \frac{x(k-i)}{\sum_{i=0}^{N-1} x^2(k-i)} \qquad for \;\; i = 0 \;\ldots\; N-1$$

$$0 < (1-\mu\alpha) < 1 \qquad (8.15)$$

There are several other algorithms available for adjusting the weights of adaptive filters including recursive least squares (RLS) and Kalman filters. Detailed discussion and derivation of adaptive filters is beyond the scope of this text. Many of these algorithms are available in MATLAB. The lab exercise at the end of this chapter introduces the adaptive filter blocks in Simulink.

Answers to Chapter 8 Challenge Questions

Question 8.1 Close examination of the plot in Figure 8.4(a) reveals a very small peak at 300 Hz prior to the much larger peak and a very small peak at 100 Hz in between the two much larger peaks. Why does this occur?

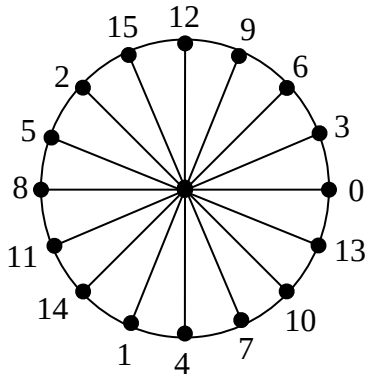
The input samples are fed into a buffer of length 256 and an FFT is computed. The buffer is then filled with 256 new input samples and a new FFT is computed. This process continues over the entire duration of the input signal. The signal in this example begins as a 100 Hz tone then switches to a 300 Hz tone. There are certain windows of time where the buffer of data contains some samples of the 100 Hz tone and some samples of the 300 Hz tone. The small peak at 300 Hz prior to the larger peak indicates that the buffer was filled mostly with samples of the 100 Hz tone but had a few samples of the 300 Hz tone. The small peak at 100 Hz between the two larger peaks at 100 Hz is indicative of a sampling time window that produced a buffer mostly filled with samples at 300 Hz with a few samples at 100 Hz. This concept is explored in more detail in Example 8.4.

Question 8.2 Example 8.4 shows that increasing the overlap of input sample values for spectral analysis results in a better reflection of how the signal spectrum varies over time. What is the disadvantage of increasing the overlap?

Increasing the overlap results in an increase in computational complexity by creating more sampling intervals and shortening the amount of time to compute each FFT. In Example 8.4, an overlap of 128 (for a 256-PT FFT) doubled the number of sampling windows and cut the time in half for computing each new FFT. Instead of waiting for 256 new input samples before computing the next FFT, the FFT is computed based on 128 new input samples and 128 past sample values. Increasing the overlap to 192 in Example 8.4 resulted in four times as many FFTs computed in ¼ the amount of time compared to no overlap.

Question 8.3 For a 16-pt DFT, assume the sixteen consecutive data samples are numbered 0, 1, 2, … 15. Sketch a diagram similar to the one in Figure 8.6 that shows how the data samples are distributed among the sixteen weights for Bin 5.

First distribute 16 weights evenly spaced around a unit circle. The oldest sample, 0, has a weight of one. For the remaining samples, start at one and move clockwise to every 5^{th} sample since we are working with Bin 5. Wrapping around the circle as necessary to use all 16 data samples results in the diagram on the following page.



Question 8.4 What are the poles associated with the Sliding DFT algorithm? What does this indicate about the stability of the algorithm?

As discussed in Chapter 4, the poles are the roots of the denominator of the transfer function. There are N−1 poles at the origin, but there is one pole at $e^{j\frac{2\pi}{N}n}$. This pole lies on the unit circle which means the algorithm is only marginally stable. The system has N zeros evenly distributed about the unit circle, and one of these theoretically cancels the marginally stable pole. However, coefficient rounding of the term $e^{j\frac{2\pi}{N}n}$ could cause the pole to migrate outside the unit circle which would cause instability of the algorithm.

Question 8.5 Why is it not possible to window the incoming data samples directly for a sliding DFT?

In a sliding DFT, we lose access to the individual data samples. The sliding DFT depends on the previous DFT calculation, the newest input value, and the oldest output value. In order to use windowing, the window values for each of the samples would have to be re-adjusted at each new time interval. This adjustment at each time interval was possible for the DFT weights because every data sample experienced the exact same change in weight (a simple rotation around the unit circle). Unfortunately, the data samples don't experience any type of uniform or consistent change when shifting through a window function.

Chapter 8 Problems

Problem 8.1 For an 8-pt DFT, draw a diagram showing all the weights for Bin 6 and the corresponding data samples assuming the data samples are numbered 0, 1, … 7 with 0 being the oldest sample.

Problem 8.2: Compute the convolution y = conv(x,h) using the overlap add method. Do the all the calculations by hand. Check your answer using MATLAB.

x = [1 –1 3 –2 4 2 1]
h = [1 2 3]

Problem 8.3: Compute the convolution y = conv(x,h) using the overlap save method. Do the all the calculations by hand. Check your answer using MATLAB.

x = [1 –1 3 –2 4 2 1]
h = [1 2 3]

Problem 8.4:

(a) Write an m-file to implement the overlap add algorithm. Inputs to the function should be the filter coefficients, h, and the input x. Output of the function should be the filter output, y.
(b) Verify that the m-file works by doing the following:
- Design an FIR filter using the Filter Design and Analysis Tool
- Define some sinusoidal input in the passband of your filter.
- Pass the input and filter coefficients to your overlap add m-file and plot the results.
- Compute y = conv(x,h) and plot the results. The two plots should match.

Problem 8.5:

(c) Write an m-file to implement the overlap save algorithm. Inputs to the function should be the filter coefficients, h, and the input x. Output of the function should be the filter output, y.
(d) Verify that the m-file works by doing the following:
- Design an FIR filter using the Filter Design and Analysis Tool
- Define some sinusoidal input in the passband of your filter.
- Pass the input and filter coefficients to your overlap add m-file and plot the results.
- Compute y = conv(x,h) and plot the results. The two plots should match.

Problem 8.6: Refer to Example 8.3 Suppose the input signal is

$$x(t) = \begin{cases} \sin(2\pi(1250)t) & 0\ ms \le t < 24\ ms \\ \sin(2\pi(725)t) & 24\ ms \le t < 42\ ms \\ \sin(2\pi(400)t) + \sin(2\pi(550)t) & 42\ ms \le t \le 60\ ms \end{cases}$$

and the input signal is sampled at Fs = 8000 Hz.

(a) Create the input signal in the MATLAB.
(b) Build the Simulink model shown in Figure 8.3 for block processing an FFT. Set the FFT length to 256 with no overlap and set the window length to 256. Run the model then plot the 3-D FFT similar to Figure 8.5. Does the FFT reflect the true spectrum of the input signal? Explain.
(c) Repeat part b for an FFT and window length of 128. How does the decrease in buffer size affect the spectrum?
(d) Repeat part c for an FFT and window length of 64.

Problem 8.7: Refer to Example 8.4. Suppose the input signal is

$$x(t) = \begin{cases} \sin(2\pi(1250)t) & 0\ ms \le t < 24\ ms \\ \sin(2\pi(725)t) & 24\ ms \le t < 42\ ms \\ \sin(2\pi(400)t) + \sin(2\pi(550)t) & 42\ ms \le t \le 60\ ms \end{cases}$$

and the input signal is sampled at Fs = 8000 Hz.

(a) Create the input signal in the MATLAB.
(b) Make a table similar to Table 8.1 and 8.2 that show the time intervals for a 128-pt FFT with no overlap, a 128-pt FFT with an overlap of 32, and a 128-pt FFT with an overlap of 64.
(c) Build the Simulink model shown in Figure 8.3 for block processing an FFT. Set the FFT length to 128 with no overlap and set the window length to 128. Run the model then plot the 3-D FFT similar to Figure 8.5. Does the FFT reflect the true spectrum of the input signal? Explain. (*Note: this is part c of Problem 8.6)*
(d) Repeat part c using an overlap of 32 for the FFT. How does the overlap affect the spectrum? Why?
(e) Repeat part c using an overlap of 64 for the FFT. How does the overlap affect the spectrum? Why?

Problem 8.8: Refer to Example 8.5. Suppose the input signal is

$$x(t) = \begin{cases} \sin(2\pi(349)t) & 0\ s \le t < 0.25 \\ \sin(2\pi(440)t) & 0.25\ s \le t < 0.5\ s \\ \sin(2\pi(349)t) + \sin(2\pi(523)t) & 0.5\ s \le t \le 1.0\ s \end{cases}$$

and the input signal is sampled at Fs = 8000 Hz.

(a) Assuming a 256-pt DFT, calculate the discrete bin frequencies (and bin numbers) closest to the frequencies in the input signal.
(b) Perform a sliding DFT on the input signal but only calculate the DFT at the bin frequencies determined in part a. Plot the results for each of the bin frequencies.
(c) Repeat part b with a windowed sliding DFT using one of the windows from Table 8.4. Compare the results with part b.

CHAPTER 8 LAB EXERCISE
Adaptive FIR Filters

Objectives

1. To experiment with adaptive FIR filters for system identification using Simulink.
2. To explore the effectiveness of adaptive FIR filters for noise cancellation.

Procedure

A. System Identification using a Normalized LMS Adaptive Filter

1. Build the Simulink system shown in Figure 1.

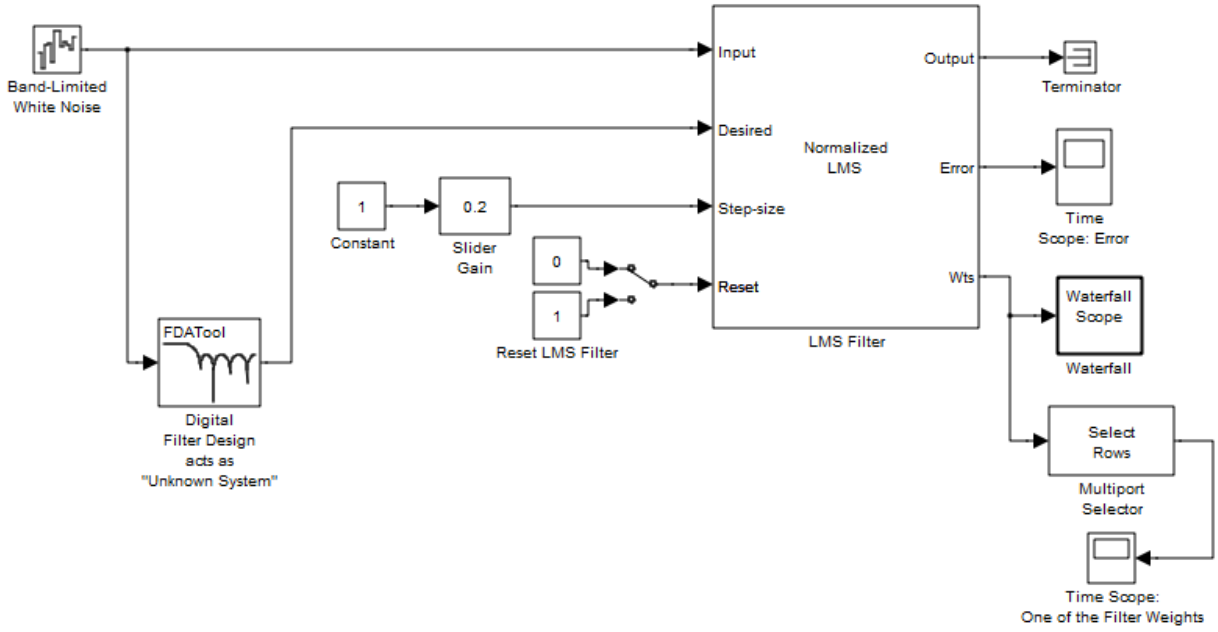


Figure 1: Simulink Model for System Identification

2. Set up the block parameters for the LMS filter block and the Digital Filter Design as shown in Figure 2.

3. Set the sample time for the band-limited white noise equal to 1/8000.

4. Set up the parameters for the Waterfall Scope as shown in Figure 3.

5. For the Multiport Selector, leave the selection as Rows and set the index to {1}. This will cause row 1 of the weights vector to be displayed on the time scope.

6. Set the stop time for the simulation to 0.1 seconds.

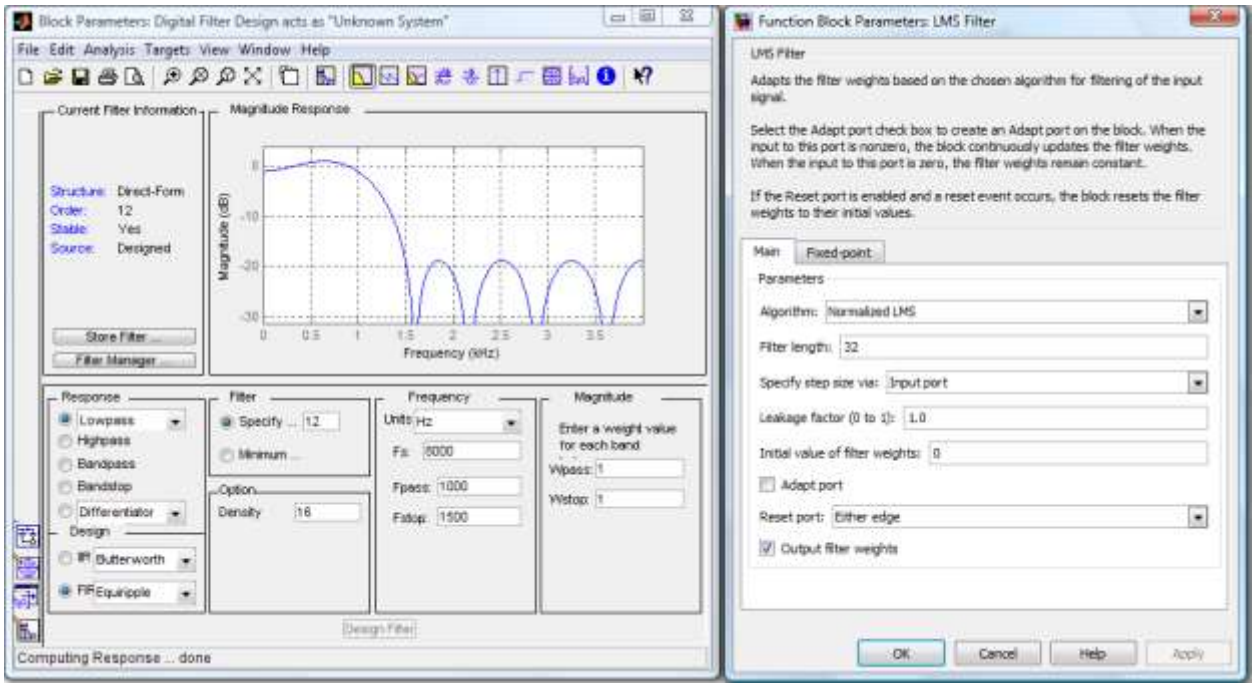


Figure 2: Block Parameters for Digital Filter and LMS Filter

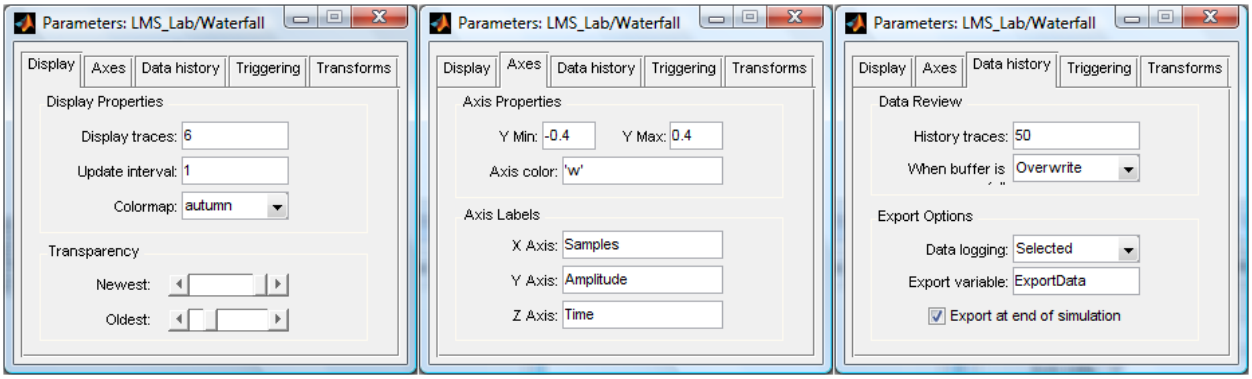


Figure 3: Block Parameters for Waterfall Scope

7. Run the simulation with the Waterfall Scope and each of the time scopes open.

   Comments:
   The error time scope should display an error that starts out relatively high but is very close to zero by the end of the simulation (0.1 seconds). The time scope for the weights simply shows how the first weight changes over time (all weights begin at zero initially). You could view any of the other weights by simply clicking on the Multiport Selector and choosing a different row. The Waterfall scope shows how all 32 weights (samples) change over time. Only the last six values (six traces) for each weight is shown. The final values for all 32 weights are available in the MATLAB workspace in a vector called ExportData.

8. Now experiment with the slider gain to change the step size of the adaptation for the filter weights. Comment on the effect of the slider gain. What happens when the adaptation step goes up to 2?

9. Set the slider gain to 0.5. Experiment with the filter length in the LMS filter block. The "Unknown System" is 12^{th} order which means it has 13 numerator coefficients. Answer the following questions:

   What happens if the filter length in the LMS filter block is shorter than the "unknown" system filter length?

   What happens if the filter length in the LMS filter block is exactly equal to the number of "unknown" filter coefficients? To really answer this question, you will need to look at the variable ExportData in the MATLAB workspace after running the simulation. Compare the LMS filter weights to the filter coefficients in the FDAT block.

   What happens if the filter length in the LMS filter block exceeds the "unknown" system filter length?

Since the system in practice is unknown, the order of the system will presumably also be unknown. What are the advantages and disadvantages of simply setting the LMS adaptive filter length really high?

10. Design a different type of "unknown" system. Choose a low-pass IIR Butterworth filter (minimum order) with a sampling frequency of 8000 Hz, pass-band 1000 Hz, stop band 2000 Hz with 40 dB of attenuation in the stop band. Can you get the LMS FIR filter to properly model the IIR "unknown" filter? How many LMS filter coefficients were needed?

11. Try a different filter type (FIR band-pass or notch) for the "unknown" system. Experiment with the adaptation step size and the LMS filter length. Comment on observations.

B. Noise Cancellation Using Adaptive Filtering

1. Build the system in Simulink shown in Figure 4.

2. Set the stop time in the model block to 90 (seconds).

3. In the ***Random Source*** block, set the Source Type to Uniform, the Maximum to 0.25, and the sample time to 1/44100.

4. In the ***From Multimedia File*** block, set the file name to songmono.wav, click the loop box, and set the number of times to repeat to 1.

5. Locate the wav file (songmono.wav) on the text website and add it to your current directory. Or, create your own songmono.wav file by saving one of your own wav files to the current directory then converting it to mono as follows:

```
[y,Fs] = audioread('nameofyourfile.ext');
ymono = y(:,1); % read the first channel of the stereo file
audiowrite('songmono.wav',ymono,Fs);
```

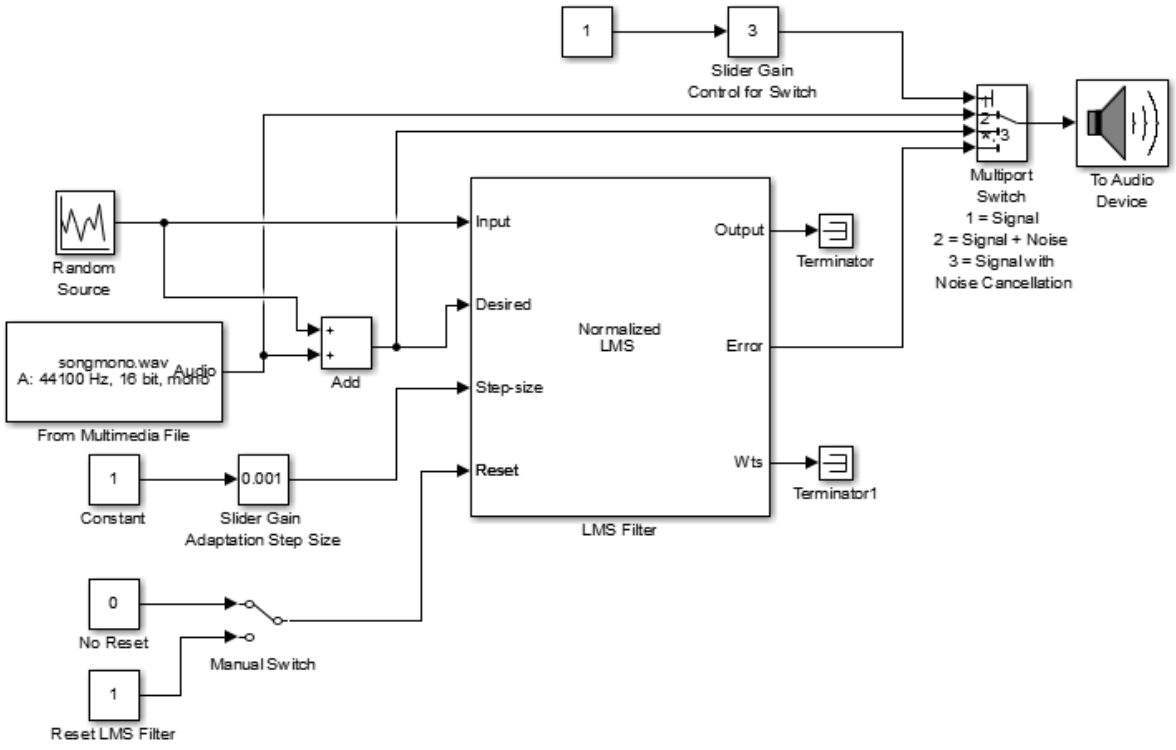


Figure 4: Noise Cancellation Model

6. On the Slider Gain Control for the switch, set the minimum to 1 and the maximum to 3. Setting this gain to 1 will cause the wav file to play. Setting the gain anywhere between 2 and 3 will cause the wav file with the noise added in to play. A gain of 3 will play the output of the LMS filter; that is the "noise reduced" wav file. Start with a slider gain of 3 (output of LMS filter).

7. The Slider Gain for the Adaptation Step Size should be set to a minimum of 0 and a maximum of 2. This slider will allow the user to adjust the adaptation step size while the simulation is running.

8. The manual switch allows the user to reset the LMS filter while the simulation is running.

9. For the LMS filter, set the algorithm to normalized LMS and the filter length to 32.

10. Run the simulation, listening to "noise reduced" or filtered signal (slider gain control switch on 3, the noisy wav file (slider gain control switch on 2 - the noise will really dominate here), and the original non-noisy signal (slider gain control switch on 1).

    Note: when you first start the simulation, the noise-reduced signal sounds noisy because the LMS filter is still modeling the noise but you should notice that noise dies off within 5 or 6 seconds .

11. Experiment with the adaptation step size.

    Does an increase in adaptation step size help or hurt in the noise reduction?

    What happens if the adaptation step size is made very small?

    How do these results compare to the effect of adaptation step size for system identification as explored in Part A?

12. Experiment with the size of the Normalized LMS filter and comment on the results.