Course Name: Digital Signal Processing Design

Course Number and Section: **14:332:447:01**

Fast Convolution Methods

Submitted by: Lance Darragh

Introduction

So far in this course we've discussed sample-by-sample processing methods of performing convolution in the time-domain. In this project we're going to explore two block-by-block processing methods known as the overlap-add and overlap-save methods of fast convolution. Both can be used to implement convolution in the time and frequency domains, but the main benefit of using these block-by-block processing methods is to take advantage of the lower computational complexity of the Fast-Fourier-Transform (FFT).

Fast-Fourier-Transform (FFT)

The FFT is a fast implementation of the DFT, which is the evaluation of the DTFT at an arbitrary number, say N, points across the frequency spectrum.

(1) $\omega_k = (2\pi k)/N, \quad 0 \leq k \leq N - 1$

In Hz, with a sampling frequency f_s, this is equivalent to:

(2) $f_k = kf_s/N, \; 0 \leq k \leq N - 1$

The N-point DFT for a length L signal is

(3) $$X(\omega_k) = \sum_{n=0}^{L-1} x(n)e^{-j\omega_k n}$$

Recall that the DTFT is periodic in ω with period 2π and that the principle domain, the Nyquist interval, is between -π and π. However, when working with the DFT, and therefore the FFT, it is conventional to define the principle domain, known as the DFT Nyquist interval, to be between 0 and 2π. Note that the negative frequency range from -π to 0 is now being observed in the interval between π and 2π, which is the same information due to the periodicity in ω. Also recall that padding zeros at the end of a signal has no effect on its DTFT.

The concept behind the FFT is to repeatedly divide the input signal block into smaller blocks whose DFT can be computed faster and then rebuild the final DFT out of these smaller DFTs. As a matter of fact, if we choose our DFT size N to be a power of 2, padding zeros if necessary, we can repeatedly divide by two until we have DFTs of a single sample to compute. We can see from equation (3) that if we simply have n = 0, the DFT of a single sample is equal to itself. For each stage in the rebuilding process, we need to multiply the values of two the N/2 DFTs together, resulting in N/2 multiplications for each stage. Because we repeatedly divided N by 2, we have $\log_2(N)$ stages, resulting in total of $(N/2)*\log_2(N)$ multiplications. This is a large advantage over the number of multiplications in computing the DFT directly, L*N, which for L =

N is N^2. There are many algorithms implementing the FFT in slightly different ways, but we'll be using MATLAB's built in function, fft.

Overlap-Add Method

In this method we divide an input signal, which may be infinite in length, into contiguous blocks of a smaller length, L. Each block of length L is filtered one at a time, and the outputs are reconstructed according to their positions in time before filtering, i.e. with the first sample of each output block lined up in time with the first sample of its input block. Recall that the linear convolution of a length L signal with an order M filter has length N = L + M, so there is a tail of M samples overlapping after the beginning of each block. So how do we manage to compute the correct convolution of the input signal if the output of each block isn't the correct length? The answer relies on the properties of linearity and time-invariance. The tail of each block, of length M, overlaps with the beginning of its subsequent block, and the correct output is obtained by adding these values together, as they would be during a linear convolution process, as shown in the following figure taken from the resources provided for this project.

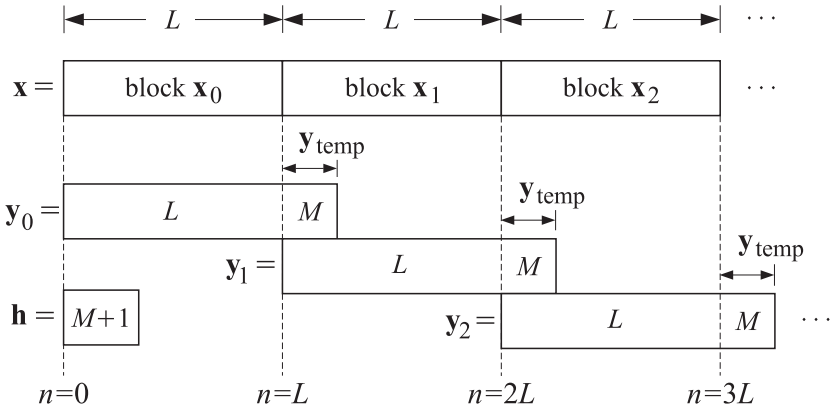


Fig. 1 Overlap-add block convolution method.

Of course, this assumes that the length of each block is greater than the filter order so that the overlap M is smaller than the block length L. Because our input is going to be broken down into blocks, we can use MATLAB's buffer function to implement this, which allows us to create a matrix with one column for each block of the input. It also allows us to pad zeros at the end of our input signal to make it a multiple of the block size. Remember, this has no effect on the DTFT, and therefore no effect on the output. Although this method does not demonstrate real-time processing, it does give a clear example of what's going in within the overlap-add method.

For demonstration, we use the following input signal and impulse response with an 8-point FFT.

```
x = 1:19; h = [1, 2, 3, 4], M = length(h) - 1; N = 8; L = M-N;          % M = 3, L = 5
```

To ensure proper overlap, we need to pad our signal with M zeros at the end.

xPad = [x(:); zeros(M,1)];

Then we can input this into the buffer function.

xMat = buffer(xPad, L)

```
xMat =

     1     6    11    16     0
     2     7    12    17     0
     3     8    13    18     0
     4     9    14    19     0
     5    10    15     0     0
```

Note that the buffer function automatically padded more zeros to ensure we have full blocks to work with. For illustration purposes, we'll transpose this so that the columns (blocks) are lined up as rows.

```
xMat =

     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19     0
     0     0     0     0     0
```

Taking the convolution of each block gives

```
yMat =

     1     4    10    20    30    34    31    20
     6    19    40    70    80    79    66    40
    11    34    70   120   130   124   101    60
    16    49   100   170   160   129    76     0
     0     0     0     0     0     0     0     0
```

Lining up the rows with an overlap of M samples gives:

```
1  4  10  20  30  34  31  20
               6  19  40  70  80  79  66  40
                           11  34  70  120  130  124  101  60
                                                16   49   100   170   160   129  76   0
```

Adding these up, we get the correct output:

y = 1 4 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 160 129 76

To implement this in the time-domain, we compute the convolution of each block (column) and store the last three (M) values in a temporary buffer. Then we'll output the five (N-M) values that are correct and iterate to the next block. Once we compute the convolution of the next block, we'll add the three values stored in the temporary buffer to the first three values of this column, which now gives us the correct output values. Then, we'll output the next five values and repeat the procedure until we run through all of the blocks of the input signal.

In the frequency domain, the process is the same except in the method of convolution used. Here, we take the FFT of the impulse response, which only needs to be done once, and point-wise multiply this by the FFT of each column in xMat and take the inverse FFT to compute the blocks of the output. The rest of the procedure is the same as above. We're going to be using the overlap-add method for the next few projects, so we've created this algorithm to work in both the time and frequency domains. The algorithm is given in Appendix A.14.

Overlap-Save Method

The overlap-save method is very similar to the overlap-add method except that it applies the properties of linearity and time invariance before filtering by allowing the input blocks to overlap by the amount of the filter order M instead of adding the overlapping portions of the resulting outputs. The only problem is that the first M samples are not computed correctly due to the reversal of the impulse response under convolution, but this can be corrected by delaying the input by M samples. In this method we choose the block length to be equal to the power-of-two FFT length, N. The following figure is from the resources given for the project.

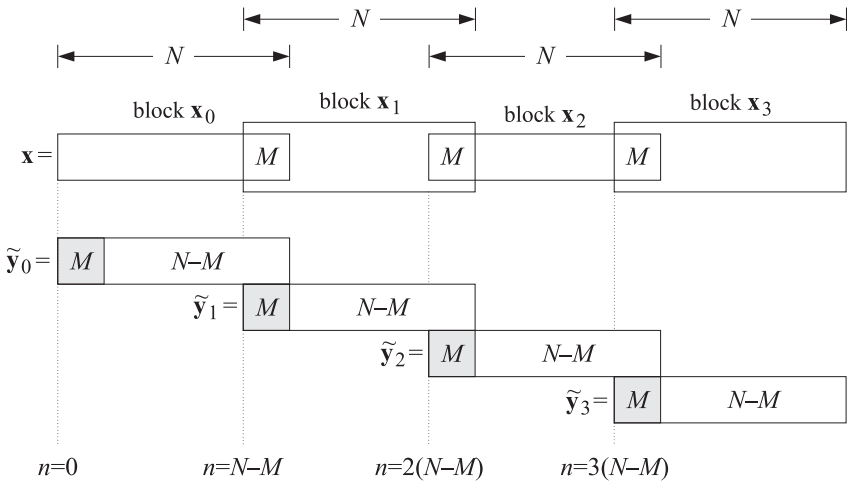


Fig. 2 Overlap-save method of fast convolution.

Using the same example as before, we construct the following matrix using the buffer function, only this time padding M zeros at the beginning. This is again the transpose of the matrix for illustration purposes.

```
xMat =

     0     0     0     1     2     3     4     5
     3     4     5     6     7     8     9    10
     8     9    10    11    12    13    14    15
    13    14    15    16    17    18    19     0
    18    19     0     0     0     0     0     0
```

Taking the convolution of each row gives the following matrix.

```
yMat =

     0     0     0     1     4    10    20    30    34    31    20
     3    10    22    40    50    60    70    80    79    66    40
     8    25    52    90   100   110   120   130   124   101    60
    13    40    82   140   150   160   170   160   129    76     0
    18    55    92   129    76     0     0     0     0     0     0
```

Now we wrap each row (column in the algorithm) modulo-N gives us:

0 0 0 1 4 10 20 30
34 31 20

3 10 22 40 50 60 70 80
79 66 40

8 25 52 90 100 110 120 130...
124 101 60

...13 40 82 140 150 160 170 160
129 76 0

18 55 92 129 76 0 0 0
0 0 0

Discarding the first 3 samples gives us the correct output:

y = 1 4 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 160 129 76

For the sake of simplifying the algorithm, given in Appendix A.15, we note that since the last three samples are wrapped to the first 3 samples, and then discarded, this is equivalent to discarding the first and last 3 samples of each block. There are times when keeping the wrapped data is important; data wrapping is a fundamental principle of the DFT. For these cases we can use the datawrap function in MATLAB.

Matched Filtering

An application of these fast convolution methods is known as matched filtering. It is commonly used to detect the presence of signals in radar processing. These signals are typically reflected versions of a known transmitted signal, and the time delay between the transmitted and received signals can be used to determine the distance to the object that reflected back the signal. The product of the time delay, t_d and the speed of light, c, will be equal to twice the distance, d, between the antenna and the object. So that $d = ct_d/2$.

Consider a radar system that transmits a signal, s(t), and receives a signal, $x_{rec}(t)$, consisting of a delayed and attenuated version of s(t) along with some noise from the environment, v(t).

(4) $x_{rec}(t) = as(t - t_d) + v(t)$

where a is the reflection coefficient.

A matched filter has an impulse response that is the reverse of the known, transmitted signal.

(5) $h(t) = s(T - t), \quad 0 \leq t \leq T$

where T denotes the maximum time value of the transmitted signal.

Let's analyze the convolution of these two signals.

(6) $$y(t) = \int_{-\infty}^{\infty} h(\tau)x_{rec}(t-\tau)d\tau$$

(7) $$y(t) = \int_{-\infty}^{\infty} h(\tau)as(t-t_d-\tau)d\tau + \int_{-\infty}^{\infty} h(\tau)v(t-\tau)d\tau$$

Letting $y_s(t)$ denote the first integral on the right hand side, and letting $y_v(t)$ denote the second integral consisting of the noise, we can define the signal-to-noise ratio (SNR) as

(8) $$SNR = \frac{E[y_s^2(t)]}{E[y_v^2(t)]}$$

(9) $$SNR \propto \frac{|\int_{-\infty}^{\infty} h(\tau)s(t-t_d-\tau)d\tau|^2}{\int_{-\infty}^{\infty} h(\tau)d\tau}$$

Representing these as vectors and using the dot product, we have

(10) $$SNR \propto \frac{|\overline{h} \cdot \overline{S_R}|^2}{\overline{h^T} \cdot \overline{h}}$$

where $\mathbf{S_R}$ denotes the reverse of s(t). Using the Schwartz inequality, it can be shown that the SNR is maximized when we have an impulse response that is equal to the reverse of the signal, that is

(11) $\mathbf{h} = \mathbf{S_R}$

This is the main benefit of using a filter whose impulse response "matches" the known impulse sent from the source. Our main goal will be to see when this signal is a maximum. Thsi will give us the time delay, t_d, that we're looking for. Because we're looking for the location of the maximum, not the value of the maximum, for analytical purposes we can disregard the actual value of the reflection coefficient for now. Plugging this impulse response into $y_s(t)$, we obtain

(12) $$y_s(t) = \int_{-\infty}^{\infty} s(T-\tau)s(t-t_d-\tau)d\tau$$

Shifting both signals by the same amount, T

(13) $$y_s(t) = \int_{-\infty}^{\infty} s(-\tau)s(t-t_d-T-\tau)d\tau$$

which is equivalent to the definition of the auto-correlation with a lag of t - t_d - T. In other words,

(14) $$y_s(t) = R_{ss}(t-t_d-T)$$

Because the autocorrelation of any function is a maximum at Rss(0), $y_s(t)$, our convolution of the signal component of the received signal and our matched filter, is a maximum at

(15) $t = t_d + T$

Directly computing the cross-correlation of the transmitted pulse and the signal component of the received signal gives us

(16) $$R_{ss}(t_d) = \int_{-\infty}^{\infty} s(t)s(t - t_d)dt$$

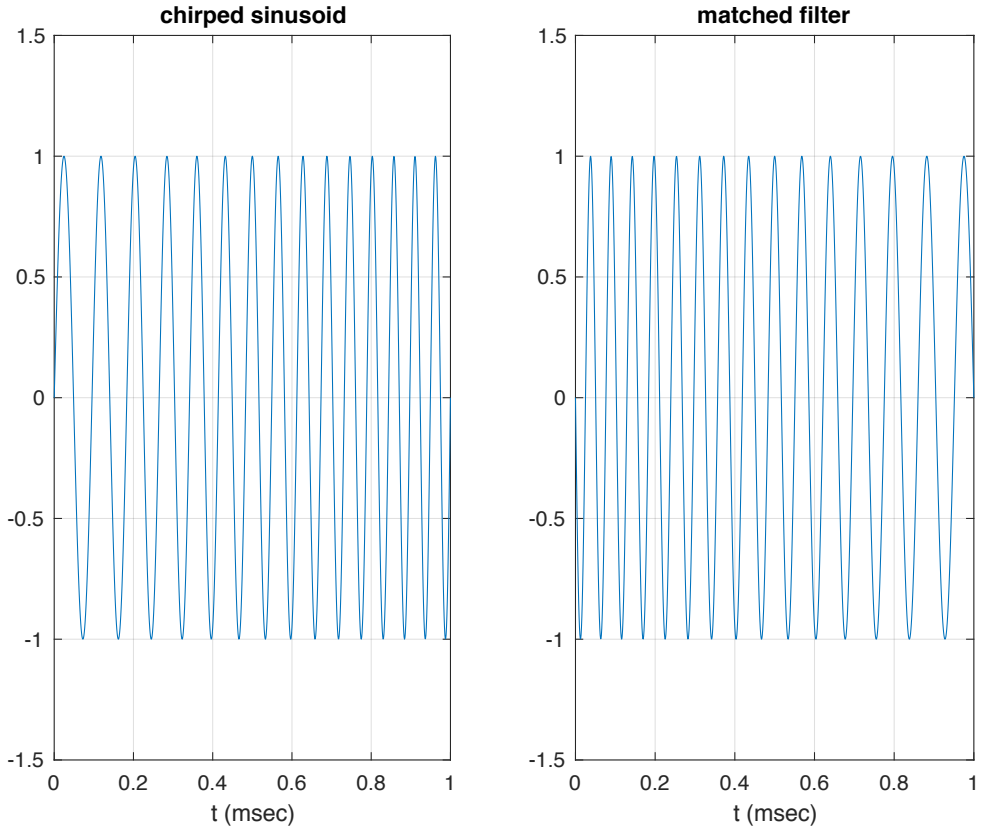
which is a maximum at $t = t_d$.

Equating the two results, we see that the actual value of the delay is more accurately found using the cross-correlation of the transmitted and received signals. However, this is far more computationally complex, being on the order of N^2, than using the matched filter with the FFT. Yet, we notice that the only difference between the computed values is a known quantity, T, the duration of the pulse signal we transmitted. So, given a known pulse signal of a fixed duration, we can reduce our computational complexity to be on the order of $(N/2)*\log_2(N)$ by simply shifting our detection results by the length of the transmitted signal, T. This can make a big difference in the resolution obtained in real-time imaging of radar processing.

As an example, let's use the following chirped sinusoid as our pulse signal.

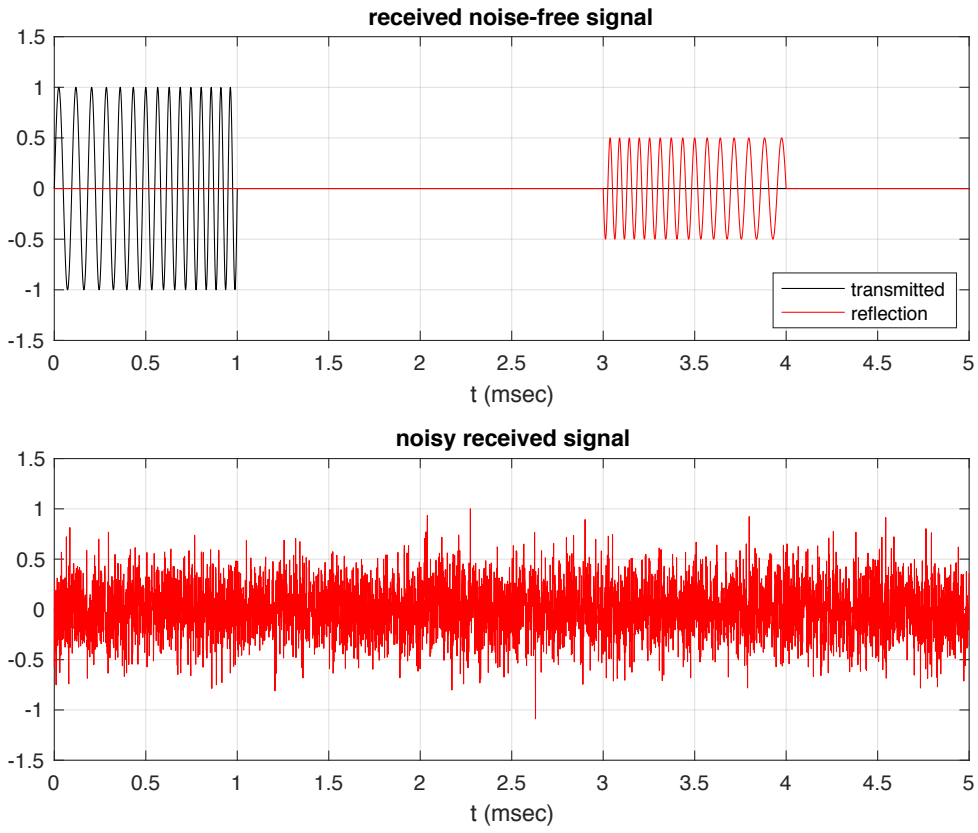
(17) $s(t) = \sin(20\pi t + 10\pi t^2), \quad 0 \le t \le 1$

with t in milliseconds. We'll sample the signal at f_s = 1MHz and construct the matched filter as h = s(T - t), where T = 1 (msec).

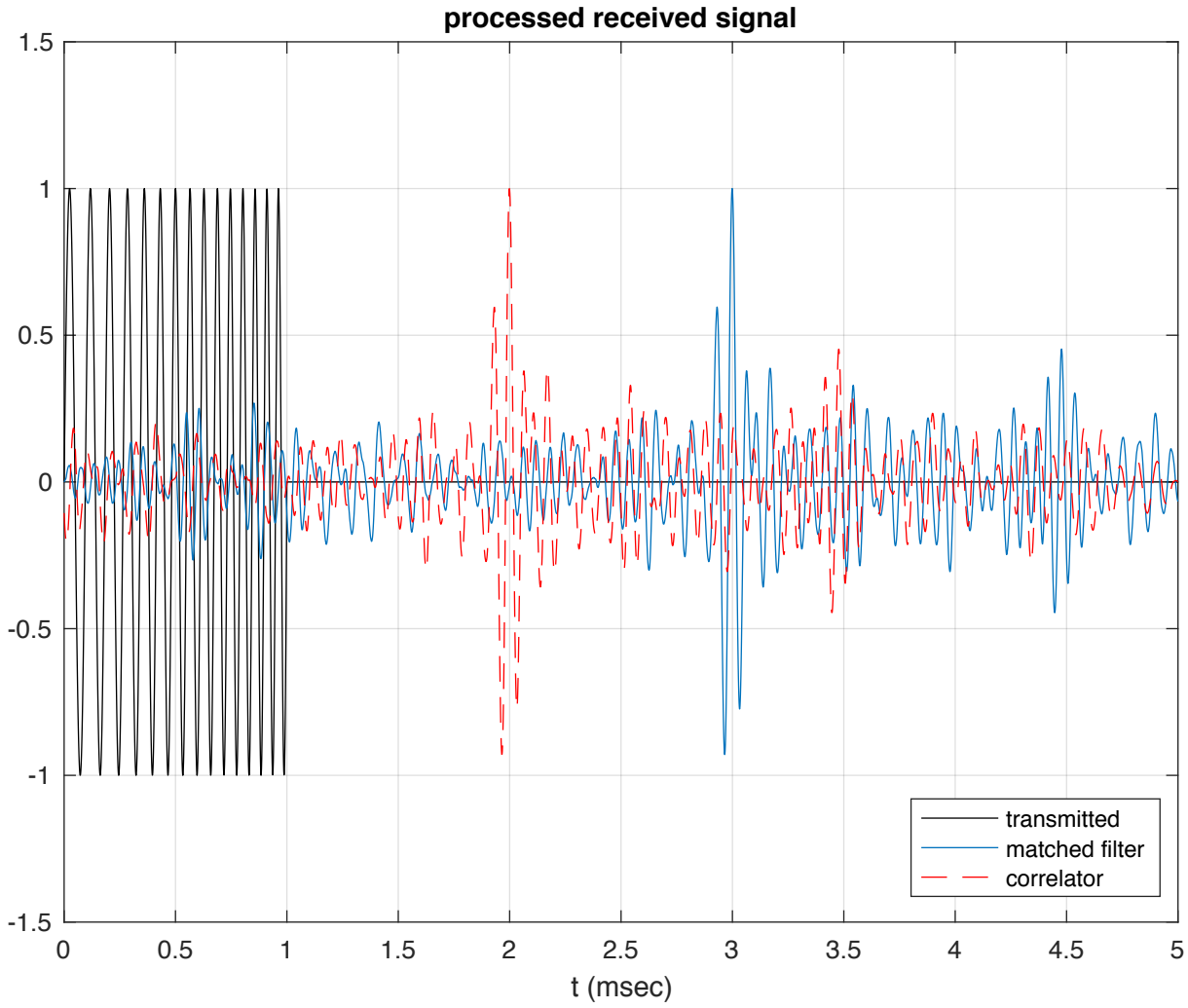


We send this pulse and record the received signal for 5 msec. It will consist of an attenuated reflection of the sent pulse along with some environmental noise. Our reflection coefficient is 0.5 and our time delay is $t_d = 3$ msec.

(18) $x_{rec}(t) = 0.5s(t-3) + v(t)$



Processing the received signal with both the cross-correlation and the matched filter obtains the results shown on the next page.



As we can see, the difference in the detected delays is exactly equal to the time duration of the transmitted signal. The MATLAB code to process these signals is given in Appendix A.16.

Appendix A.14: Overlap-Add Method of Fast Convolution

```
function y = ovadd(h,x,N,type)
% Applies the overlap-add method of fast convolution to signal x
% h = impulse response
% N = nearest power of 2 above the length of h
% type = 't' for time-domain, or 'f' for frequency-domain

n = 1;                  % Counter variable for the number of output
 samples
M = length(h) - 1;      % Amount of overlap
L = N - M;              % Length of each block size

xPad = [x(:); zeros(M,1)];
xMat = buffer(xPad,L);      % Create matrix of nonoverlapping blocks
[l,m] = size(xMat);     % To determine the number of blocks, m

ytemp = zeros(M,1);     % Temporary buffer for convolution tails

if type == 't'          % Time-domain convolution

for j = 1:m
    y1 = conv(h,xMat(:,j)); % Convolve each block

    for k = 1:M
        y1(k) = y1(k) + ytemp(k);   % Add blocks with M overlaps
        ytemp(k) = y1(k+L);         % Input the overlaps for the next
 block
    end

    for k = 1:L
        if n <= (length(x)+M)       % Output correct number of samples
            y((j-1)*L + k) = y1(k);         % Output correct values
            n = n + 1;
        end
    end


end

elseif type == 'f'     % Frequency-domain convolution

    H = fft(h(:),N);    % FFT of impulse response

for j = 1:m
    X = fft(xMat(:,j),N);   % FFT of each block
    y1 = real(ifft(H.*X));  % output of convolution for each block

    for k = 1:M
        y1(k) = y1(k) + ytemp(k);   % Add blocks with M overlaps
```

Appendix A.15: Overlap-Save Method of Fast Convolution

```
function y = ovsave(h,x,N,type)
% Applies the overlap-save method of fast convolution to signal x
% h = impulse response
% N = nearest power of 2 above the length of h
% type = 't' for time-domain, or 'f' for frequency-domain


n = 1;              % Counter variable for the number of output
 samples
M = length(h) - 1;      % Amount of overlap
L = N - M;              % To output the correct number of samples per
 block

xPad = [x(:); zeros(M,1)];  % Pad zeros for the correct number of
 blocks
xMat = buffer(xPad,N,M);    % Create matrix of overlapping blocks
[l,m] = size(xMat);         % To determine the number of blocks, m

if type == 't'          % Time-domain convolution

for j = 1:m
    y1 = conv(h,xMat(:,j));
    % Equivalent to wrapping modulo N and discarding the wrapped
 samples,
    % just less operations
    y1(1:M) = [];                %Discard first M samples
    y1(end-M+1:end) = [];        %Discard last M samples

    for k = 1:L
        if n <= (length(x)+M)       % Output correct number of samples
            y((j-1)*L + k) = y1(k);         % Output correct values
            n = n + 1;
        end
    end
end

elseif type == 'f'

    H = fft(h(:),N);    % FFT of impulse response

for j = 1:m
    X = fft(xMat(:,j),N);   % FFT of each block
    y1 = real(ifft(H.*X));  % output of convolution for each block
    % Equivalent to wrapping modulo N and discarding the wrapped
 samples,
    % just less operations
    y1(1:M) = [];           %Discard first M samples
```

Table of Contents

Appendix A.16: Overlap-add & Overlap-save Fast Convolution Methods

```
x = 1:19; % Input signal

h = [1 2 3 4]; % Filter

M = length(h) - 1; % Order of filter

N = 2^ceil(log2(2*M + 1)); % N is the nearest power of 2 above 2*M

% Using MATLAB's conv function
yconv = conv(h,x)

% Compute outputs
yovaddT = ovadd(h,x,N,'t')
yovaddF = ovadd(h,x,N,'f')
yovsaveT = ovsave(h,x,N,'t')
yovsaveF = ovsave(h,x,N,'f')
```

Applications: Matched Filter vs. Cross-Correlation

```
clear, clc
load xrec;
fs = 1e6; Ts = 1/fs;            % Sampling rate and period
t = 0:Ts*1e3:1;                 % Duration of pulse
s = @(t) sin(20*pi*t + 10*pi*t.^2); % Pulse signal
T = 1;                          % Max time value of pulse
h = s(T-t);                     % Matched filter impulse response
M = length(h) - 1;              % Order of matched filter
N = 2^ceil(log2(2*M + 1));      % N is the nearest power of 2 above
 2*M

ymatch = ovsave(h,xrec,N,'t');          % Matched filter output
ymatch(length(xrec)+1:end) = [];        % Truncate convolution tail
ymatch = ymatch/max(ymatch);

t5 = 0:Ts*1e3:5;                        % Duration of received signal
s5 = [s(t),zeros(1,4000)];   % Extend pulse signal length to match
 received signal length
```

```
xCorr = xcorr(xrec,s5);     % Compute the cross-correlation between
 the sent pulse and the received signal
xCorr(1:5000) = [];% Eliminate negative portion of xCorr to only look
 at positive values of delay
xCorr = xCorr/max(xCorr);

close all
plot(t5,s5,'k',t5,ymatch,t5,xCorr,'r--'),legend('transmitted','matched
 filter','correlator')
axis([0,5,-1.5,1.5]);title('processed received signal'),xlabel('t
 (msec)'), grid on
```

Plot the Results

```
close all
subplot(1,2,1)
plot(t,s(t)), axis([0,1,-1.5,1.5]), grid on, xlabel('t (msec)'),
title('chirped sinusoid')
subplot(1,2,2)
plot(t,h),axis([0,1,-1.5,1.5]), grid on, xlabel('t (msec)'),
title('matched filter')


close all
as = 0.5*s(t - 3);
as5 = [zeros(1,3000),as,zeros(1,1000)];
xrec = xrec/max(xrec);

subplot(2,1,1)
plot(t5,s5,'k',t5,as5,'r'),legend('transmitted','reflection')
axis([0,5,-1.5,1.5]);title('received noise-free signal'),xlabel('t
 (msec)'), grid on
subplot(2,1,2)
plot(t5,xrec,'r'),
axis([0,5,-1.5,1.5]);title('noisy received signal'),xlabel('t
 (msec)'), grid on
```

Published with MATLAB® R2017b

```
    for k = 1:L
        if n <= (length(x)+M)       % Output correct number of samples
            y((j-1)*L + k) = y1(k);         % Output correct values
            n = n + 1;
        end
    end
end
end
end
```

Published with MATLAB® R2017b

```
        ytemp(k) = y1(k+L);          % Input the overlaps for the next
 block
    end

    for k = 1:L
        if n <= (length(x)+M)        % Output correct number of samples
            y((j-1)*L + k) = y1(k);          % Output correct values
            n = n + 1;
        end
    end

end
end
end
```

Published with MATLAB® R2017b