Jacobs University Bremen

CO-522-B Intersession 2022

Communication Basics Lab

Lab Write-Up 3 : Delay and FIR

Kuranage Roche Rayan Nimnath Ranasinghe

Conducted on : 17/01/2022

A short explanation of the difference between a usual MATLAB simulation and real-time DSP code. Also, briefly explain why buffer-oriented processing is needed and what affects the latency and throughput of an algorithm.

The differences between a real-time DSP code and a MATLAB simulation are as follows,

- In a real-time DSP code, the receiving and processing is done simultaneously, i.e. the incoming data stream is continuous and hence, the data must be processed continuously since all the data is NOT available to process at a given time.

- For a real-time DSP code, keeping track of the time (i.e. synchronisation) is very important since the code must NOT have a long delay (the time between receiving a stream of data and processing it to give an output, also known as latency). However, in a MATLAB program, since all the data is available immediately to process, the timing, while still important, would not affect the desired capability of the algorithm by a large amount.

Buffer-oriented processing is a consequence of the first point mentioned above. In a real-time DSP code, since all the samples are not available at a given time, the incoming data stream (bit stream) is partitioned into a buffer containing ‘N’ samples (where N is the size of the buffer, i.e. the number of samples in the buffer) to generate ‘N’ output samples. The information of previous buffers is stored in state for the following buffers to be processed correctly.
The size of the buffer (N) is directly proportional to the latency in the code. Hence, a smaller buffer size leads to a lower latency since only a smaller number of samples need to be processed at a time.
However, processing many small buffers would also lead to a wastage of resources like time in storing and restoring the state, which in turn leads to *lower overall throughput.*
To put this all together, the number of buffers is inversely proportional to the throughput, hence, when a larger number of buffers are used in a program (to try and reduce latency), there is a greater wastage of resources like time which decreases our overall throughput.
Due to these reasons, to find a balance between these two factors, most DSP programs contain constraints on which the optimal value of ‘N’ depends on.
It should also be noted that coding DSP blocks on MATLAB helps with faster processing time because of the user-friendly environment.

A diagram showing conceptually how your Delay uses a circular buffer to implement the Delay.

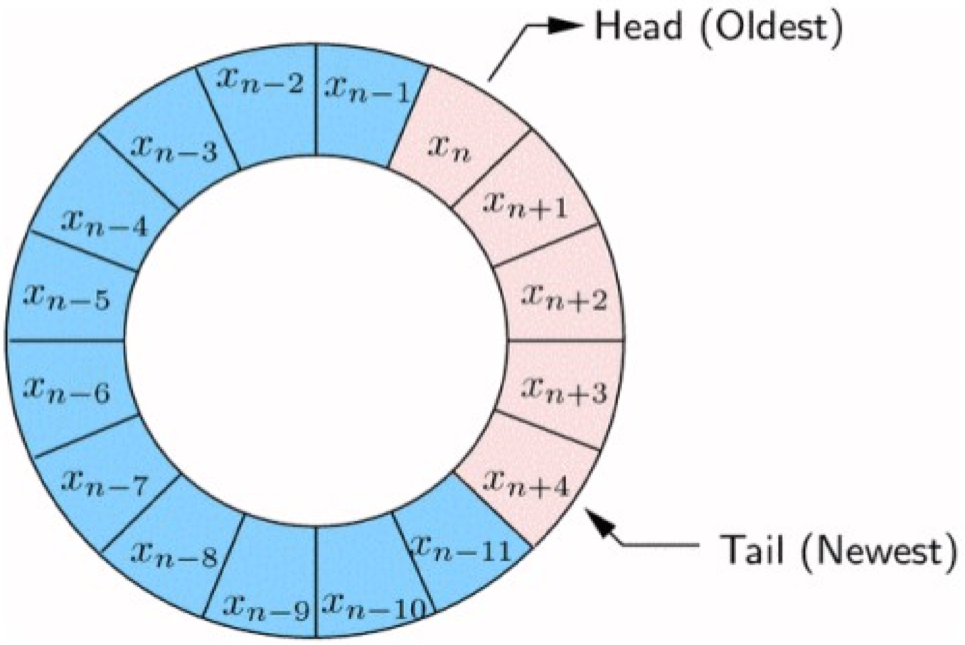


Figure: Circular Buffer for DSP

We can simulate the continuous processing of a stream of data samples with a finite buffer size using a circular buffer. As illustrated above, memory elements are arranged in a circle where an incoming data sample is placed at the tail of the buffer which results in the incrementation of the tail pointer. The oldest item which is not processed is taken from the head of the buffer to be processed after which the head pointer is incremented (A circular buffer is also known as a FIFO (First In First Out) buffer due to its arrangement, i.e. the first sample to get into the buffer is the first sample to be processed). This method is convenient because the unnecessary process of copying data is avoided. In a processor, the circular buffer is arranged in a logical, linear manner as shown below:

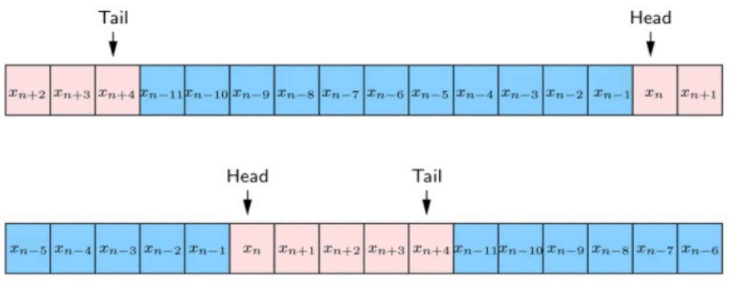

Tail
Head
x_{n+2} x_{n+3} x_{n+4} x_{n-11} x_{n-10} x_{n-9} x_{n-8} x_{n-7} x_{n-6} x_{n-5} x_{n-4} x_{n-3} x_{n-2} x_{n-1} x_n x_{n+1}
Head
Tail
x_{n-5} x_{n-4} x_{n-3} x_{n-2} x_{n-1} x_n x_{n+1} x_{n+2} x_{n+3} x_{n+4} x_{n-11} x_{n-10} x_{n-9} x_{n-8} x_{n-7} x_{n-6}

Figure: Flattened Buffer

A printout of the MATLAB code that implements your Delay with comments.

Our MATLAB code was divided into three parts as follows:

- A file containing the functions for storing the parameters and initialization of the block (delay_init.m)
- A file containing the functions for buffer processing of input items to generate a buffer of outputs (delay.m)
- And finally, two test files containing testing code to simulate buffer-oriented processing (test_delay1.m), and for the implementation of cascaded delay blocks (test_delay2.m)

MATLAB Code for delay_init.m

```
%[name]_init() — A function used to store parameters and initialize the block.
%[name]() — A function that processes a buffer of input items to generate a buffer of outputs.

function [state] = delay_init(Nmax, N);


% [state] = delay_init(Nmax, N);
%
% Initializes a delay block.
%
% Inputs:
% Nmax Maximum delay supported by this block.
% N Initial delay
% Outputs:
% state State of block
% Notes:
% For this block to operate correctly,
% you should not pass in more than Nmax
% samples at a time.

state.Nmax = Nmax;
% Store initial desired delay.
state.N = N;
```

Make the size of the buffer at least twice of the maximum delay. Allows us to copy in and then read out in just two steps.

```
state.M = 2^(ceil(log2(Nmax))+1);
% Get mask allowing us to wrap index easily
state.Mmask = state.M-1;
% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);
% Set initial head and tail of buffer
state.n_h = 0;
state.n_t = N;
```

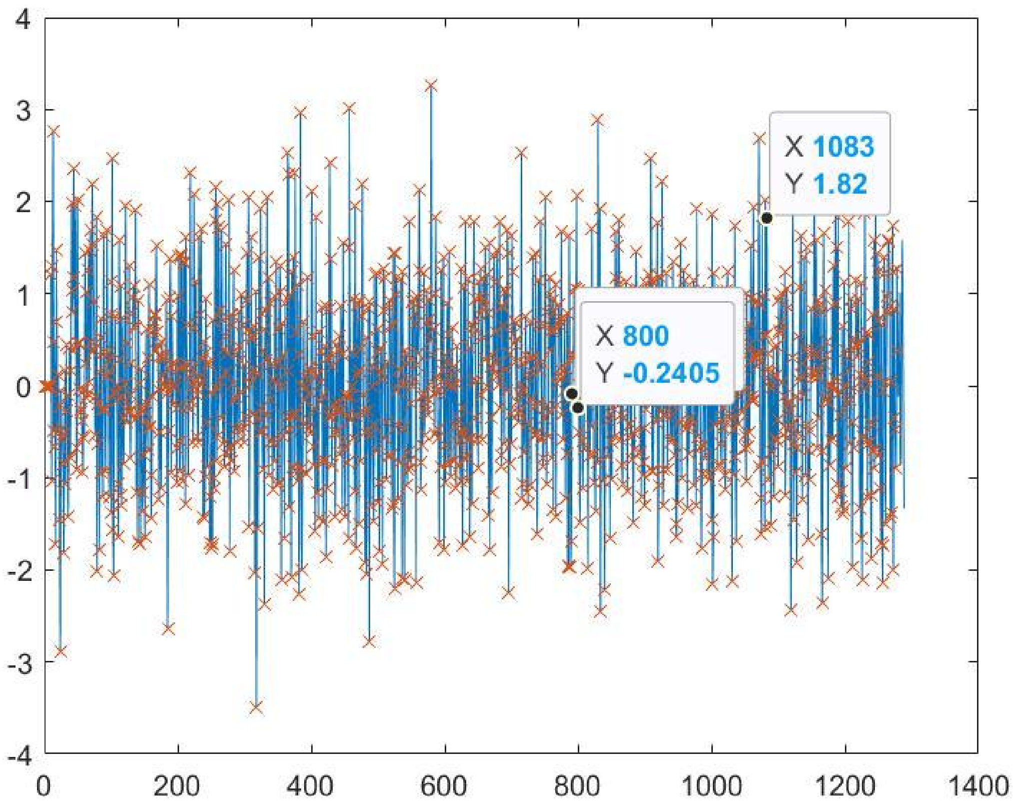
MATLAB code for delay.m

```
function [state_out, y] = delay(state_in, x);
% [state_out, y] = delay(state_in, x);
%
% Delays a signal by the specified number of samples.
%
% Inputs:
% state_in Input state
% x Input buffer of samples
% Outputs:
% state_out Output state
% y Output buffer of samples
% Get input state
s = state_in;
% Copy in samples at tail
for ii=0:length(x)-1,
 % Store a sample
 s.buff(s.n_t+1) = x(ii+1);
 % Increment head index (circular)
 s.n_t = bitand(s.n_t+1, s.Mmask);
end
% Get samples out from head
y = zeros(size(x));
for ii=0:length(y)-1,
 % Get a sample
 y(ii+1) = s.buff(s.n_h+1);
 % Increment tail index
 s.n_h = bitand(s.n_h+1, s.Mmask);
end
% Output the updated state
state_out = s;
```

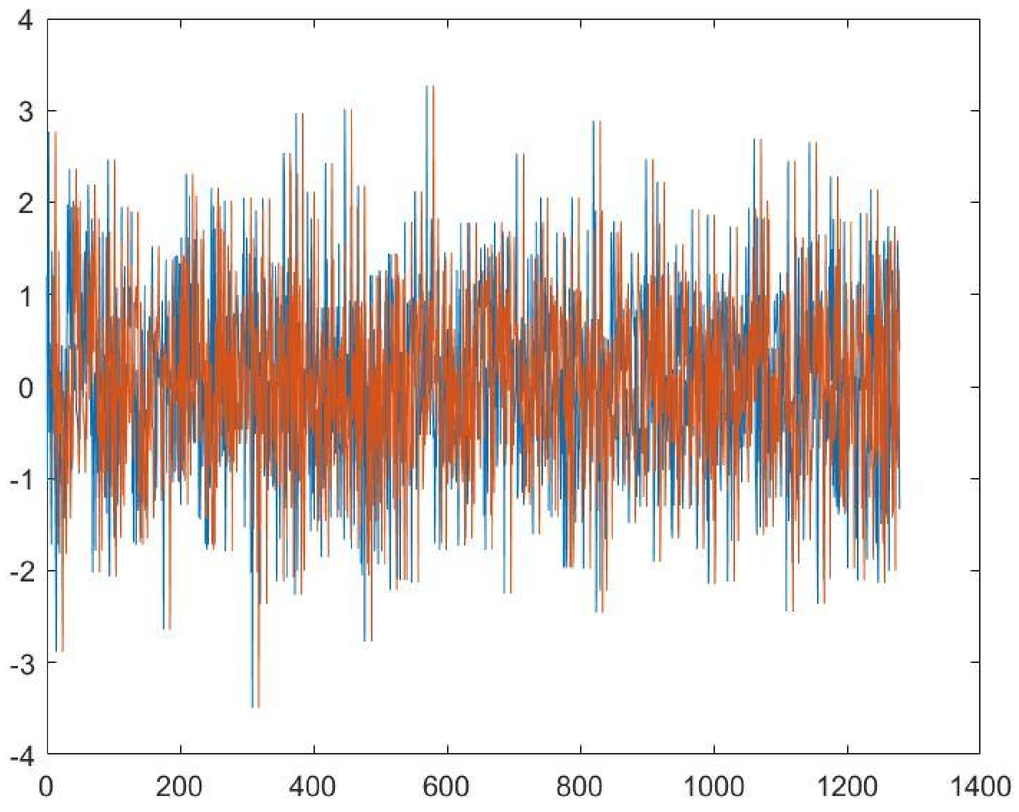
MATLAB code for test_delay1.m

```
% test_delay1.m
%
% Script to test the delay block. Set up to model the way
% samples would be processed in a DSP program.
% Global parameters
Nb = 10; % Number of buffers
Ns = 128; % Samples in each buffer
Nmax = 200; % Maximum delay
Nd = 10; % Delay of block
% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);
% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);
% Output samples
yb = zeros(Ns, Nb);
% Process each buffer
for bi=1:Nb,
 [state_delay1 yb(:,bi)] = delay(state_delay1, xb(:,bi));
end
% Convert individual buffers back into a contiguous signal.
y = reshape(yb, Ns*Nb, 1);
% Check if it worked right
n = [0:length(x)-1];
figure(1);
plot(n, x, n, y);
figure(2);
plot(n+Nd, x, n, y, 'x');
% Do a check and give a warning if it is not right. Skip first buffer
% in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk)),
 warning('A mismatch was encountered.');
End
```

The Plots Generated for this test file are as follows:



This plot shows the input and output vectors for a delay of 10

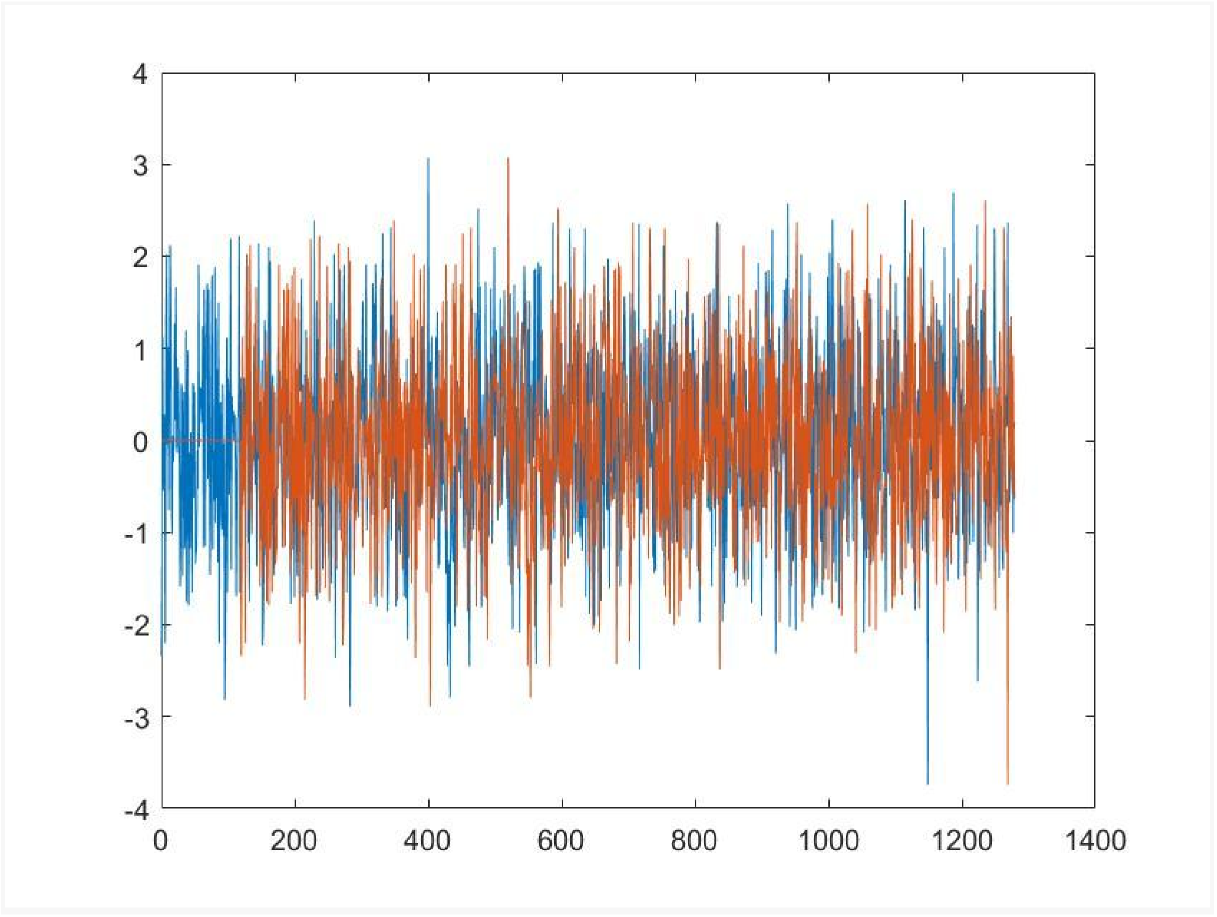


This plot shows the same sample with our shift ‘undone’.

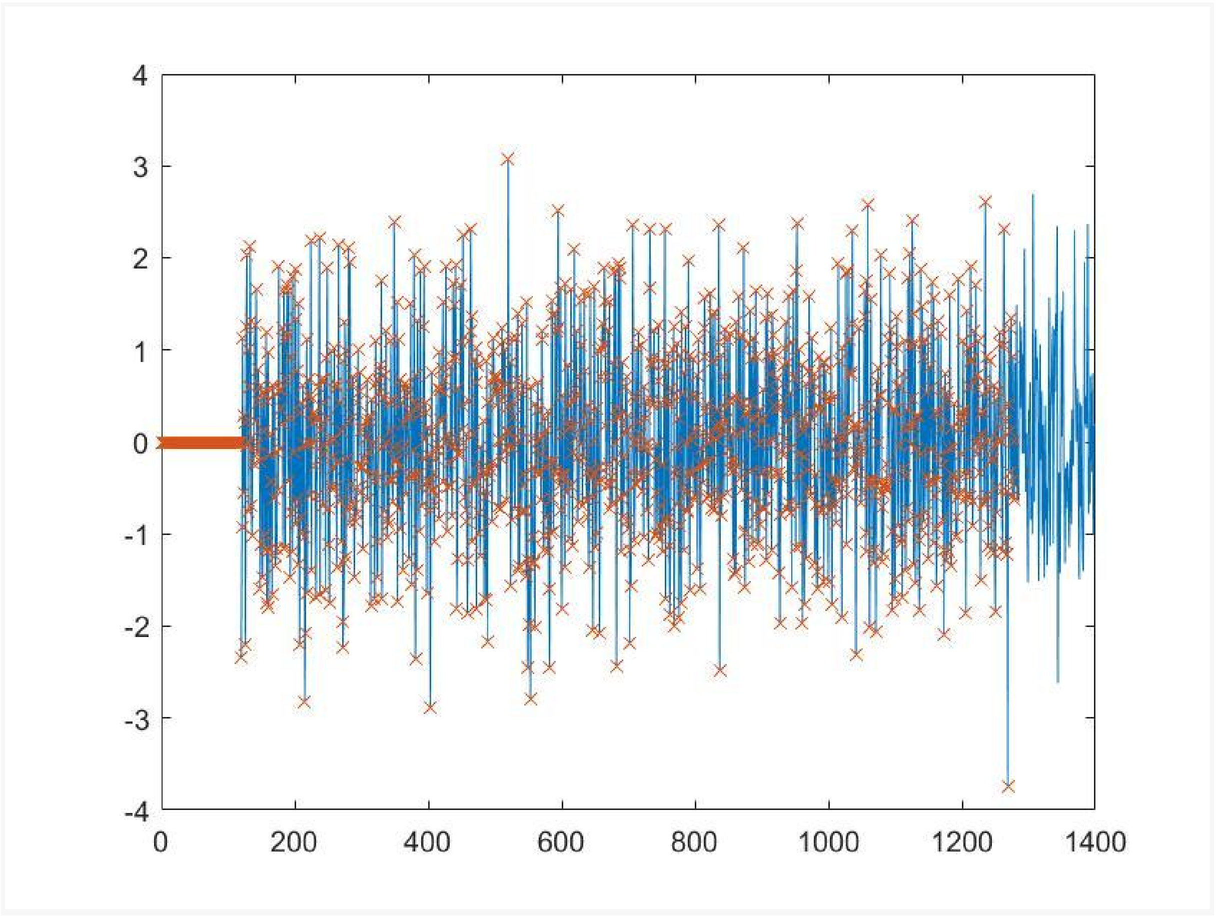
MATLAB code for test_delay2.m

```
% test_delay2.m
%
% Script to test two delay blocks. Set up to model the way
% samples would be processed in a DSP program.
% Global parameters
Nb = 10; % Number of buffers
Ns = 128; % Samples in each buffer
Nmax = 200; % Maximum delay

Nd1 = 50; % Delay of the block 1
Nd2 = 70; % Delay of the block 2

% Initialize the delay blocks
state_delay1 = delay_init(Nmax, Nd1);
state_delay2 = delay_init(Nmax, Nd2);

% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);
% Output samples
yb1 = zeros(Ns, Nb);
yb2 = zeros(Ns, Nb);
% Process each buffer in each block seperately

for bi=1:Nb,
 [state_delay1 yb1(:,bi)] = delay(state_delay1, xb(:,bi));
 % we update the value twice to compensate for 2 buffers
 [state_delay2 yb2(:,bi)] = delay(state_delay2, yb1(:,bi));
end


% Convert the new buffer back into a contiguous signal.
y = reshape(yb2, Ns*Nb, 1);
% Check if it worked right
n = [0:length(x)-1];
figure(1);
plot(n, x, n, y);
figure(2);
plot(n+Nd1+Nd2, x, n, y, 'x');
% Do a check and give a warning if it is not right. Skip first buffer
% in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd1 - Nd2) ~= y(n_chk)),
 warning('A mismatch was encountered.');
end
```

The Plots Generated for this test file are as follows:



This plot shows the input and output vectors for a cascaded delay of 120 (50 + 70)



This plot shows the sample with ‘undone’ shift for cascaded delay of 120.

For FIR lab,

A diagram showing conceptually how your FIR filter uses a circular buffer to implement the FIR equation.

Our FIR uses a circular buffer implementation as follows:

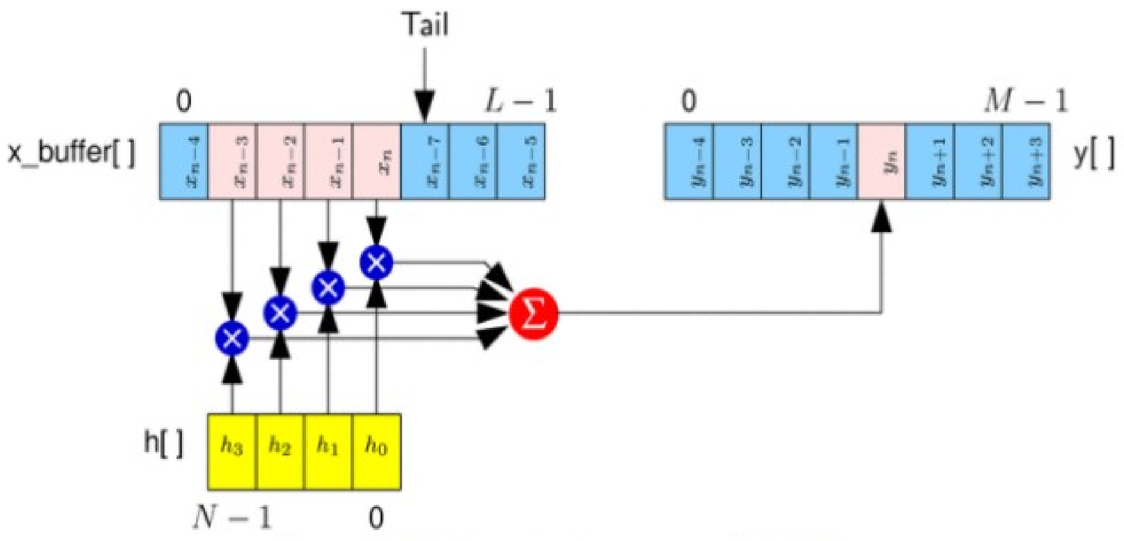


Figure: FIR filter employing a circular buffer

A printout of the MATLAB code that implements your FIR filter with comments.

Similarly to our previously implemented delay block, our MATLAB code was divided into three parts as follows:

- A file containing the functions for storing the parameters and initialization of the block (fir_init.m)
- A file containing the functions for buffer processing of input items to generate a buffer of outputs (fir.m)
- And finally, a test file containing testing code to simulate buffer-oriented processing (test_fir1.m)

MATLAB code for fir_init.m

```
function [state] = fir_init(h, Ns);



% [state] = fir_init(h, Ns);
%
% Creates a new FIR filter.
%
% Inputs:
% h Filter taps
% Ns Number of samples processed per block
% Outputs:
% state Initial state
```

Make buffer big enough to hold Ns+Nh coefficients. Make it an integer power of 2 so we can do simple circular indexing.

```
state.M = 2^(ceil(log2(state.Ns+1)));
% Get mask allowing us to wrap index easily
state.Mmask = state.M-1;
% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);
% Set initial tail pointer and temp pointer (see pseudocode)
state.n_t = Ns;
state.n_p = state.n_t - 1;
```

MATLAB code for fir.m

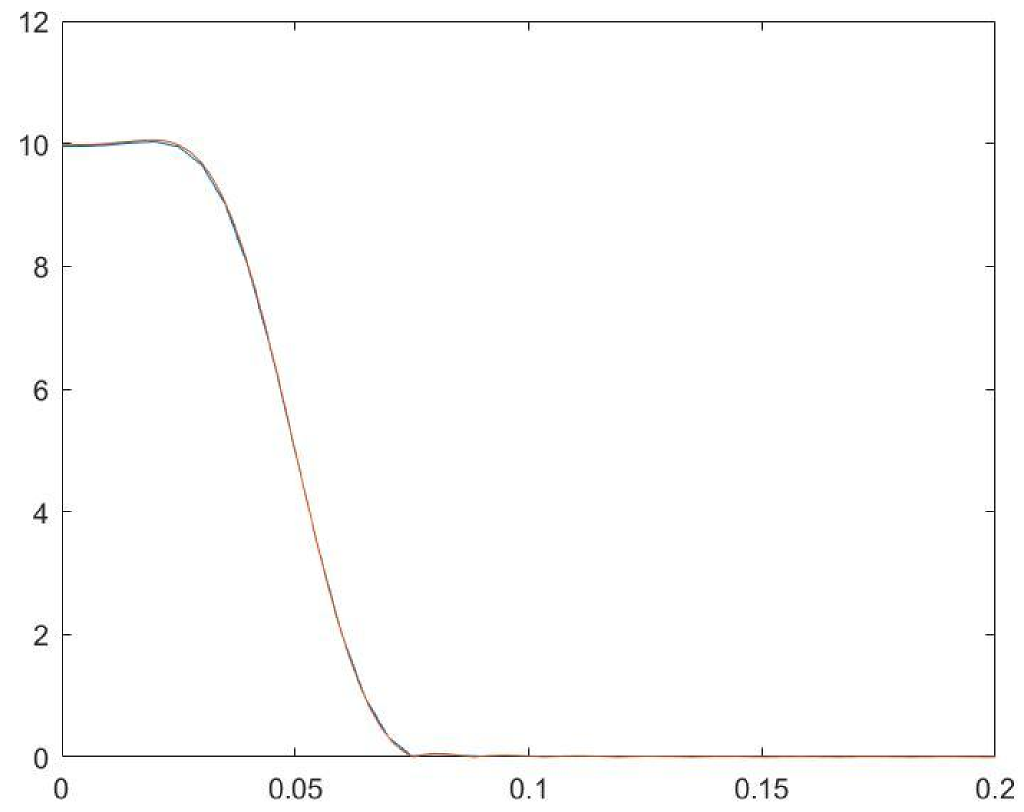
```
function [state_out, y] = fir(state_in, x);
% [state_out, y] = fir(state_in, x);
%
% Executes the FIR block.
%
% Inputs:
% state_in Input state
% x Samples to process
% Outputs:
% state_out Output state
% y Processed samples
% Get state
s = state_in;
% Move samples into tail of buffer
for ii = 0: s.Ns - 1
% Store a sample
s.buff(s.n_t+1) = x(ii+1);
% Increment head index (circular)
s.n_t = bitand(s.n_t+1, s.Mmask);
s.ptr = bitand(s.n_t+s.Mmask, s.Mmask);
sum = 0.0;
for j = 0:length(s.h)-1
sum = sum + s.buff(s.ptr+1)*s.h(j+1);
s.ptr = bitand(s.ptr+s.Mmask, s.Mmask);
end
y(ii+1) = sum;
end
% Filter samples and move into output
% Return updated state

% Copy in samples at tail


state_out = s;
```

MATLAB code for test_fir.m

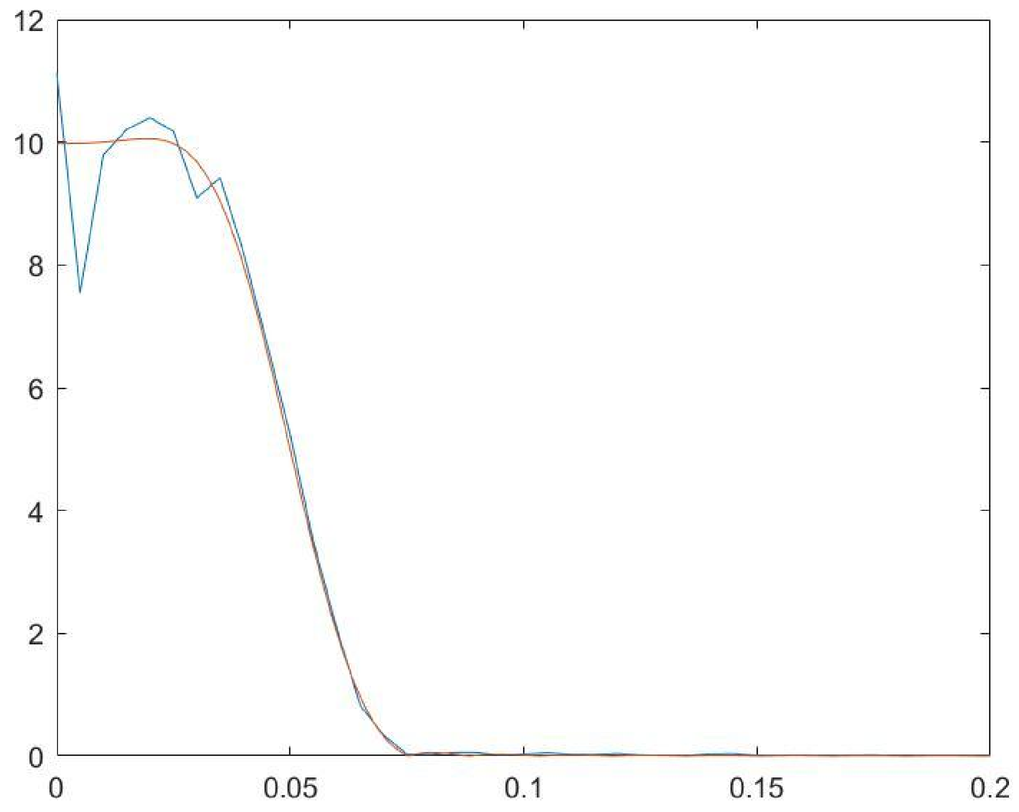
```
% test_fir1.m
%
% Script to test the FIR filter.
% Global parameters
Nb = 100; % Number of buffers
Ns = 128; % Samples in each buffer
% Generate filter coefficients
p.beta = 0.5;
p.fs = 0.1;
p.root = 0; % 0=rc 1=root rc
M = 64;
[h f H Hi] = win_method('rc_filt', p, 0.2, 1, M, 0);
% Generate some random samples.
x = randn(Ns*Nb, 1);
% Type of simulation
stype = 1; % Do simple convolution
%stype = 1; % DSP-like filter
if stype==0,
 y = conv(x, h);
elseif stype==1,
 % Simulate realistic DSP filter

 % ADD YOUR CODE HERE !!!

 state_fir1 = fir_init(h,Ns);
 % Reshape into buffers
 xb = reshape(x, Ns, Nb);
 % Output samples
 yb = zeros(Ns, Nb);
 % Process each buffer
 for bi=1:Nb
    [state_fir1 yb(:,bi)] = fir(state_fir1, xb(:,bi));
 end
% Convert individual buffers back into a contiguous signal.
y = reshape(yb, Ns*Nb, 1);



else
 error('Invalid simulation type.');
end
% Compute approximate transfer function using PSD
Npsd = 200; % Blocksize (# of freq) for PSD
[Y1 f1] = periodogram(y, [], Npsd, 1);
[X1 f1] = periodogram(x, [], Npsd, 1);
plot(f1, abs(sqrt(Y1./X1)), f, abs(H));
xlim([0 0.2]);
```

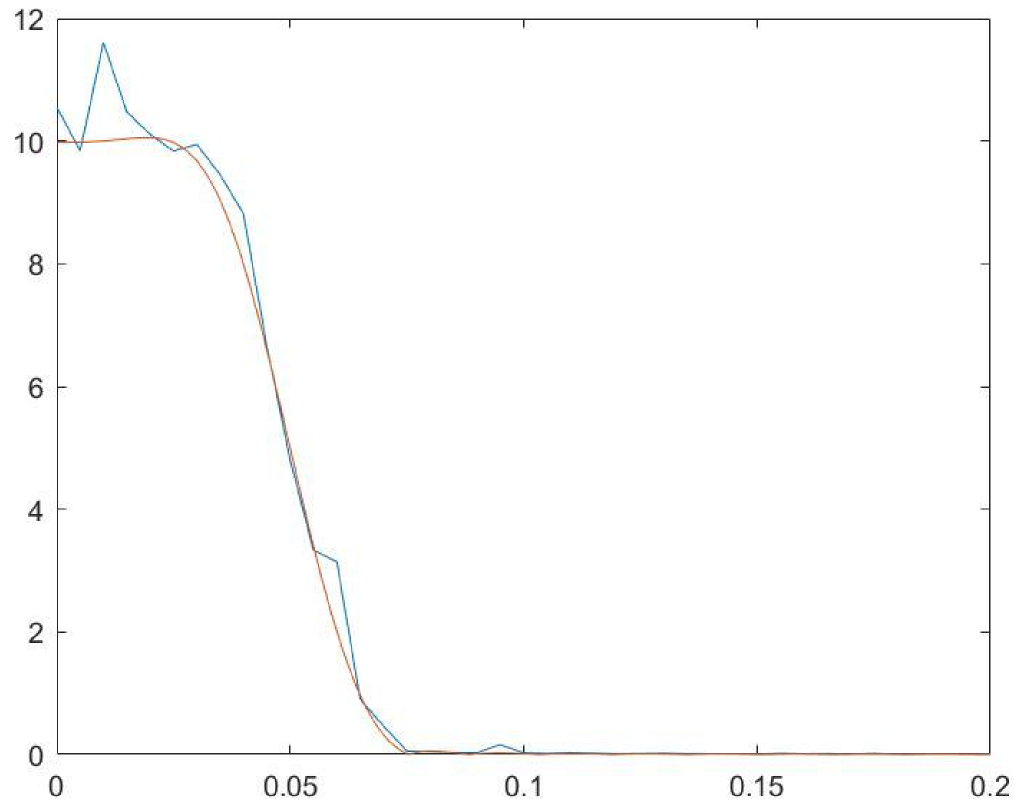
Plots showing the ideal response of your filter compared to the simulated response of the filter. Explain any discrepancies.

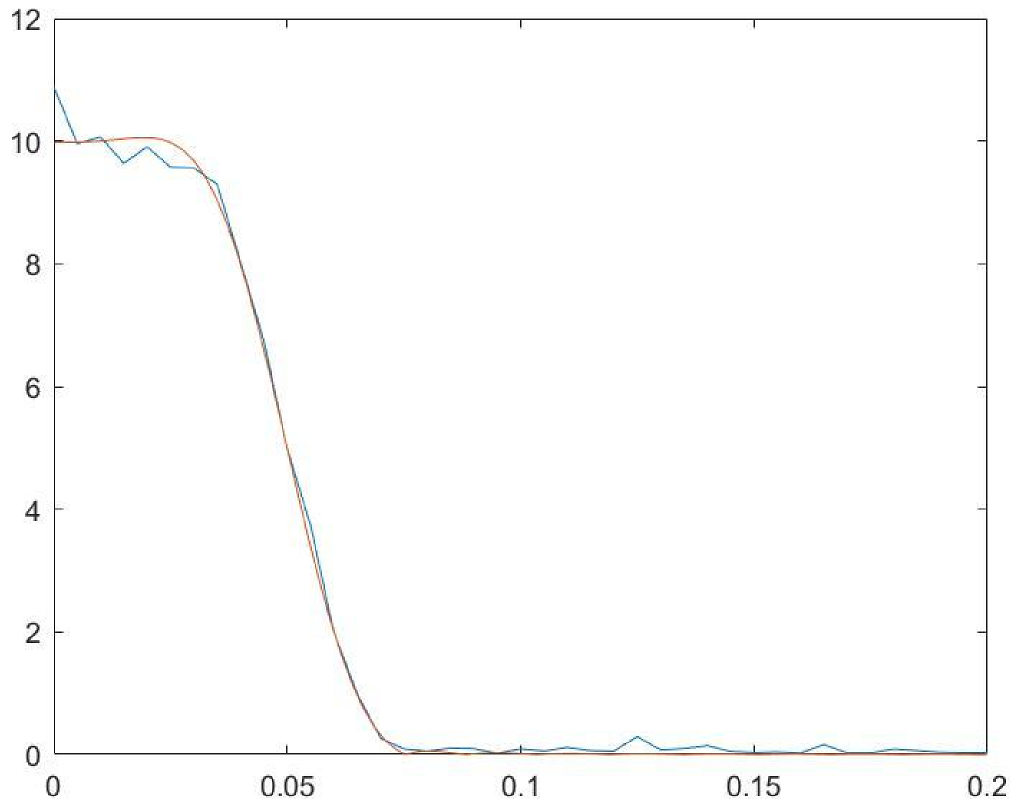
- The plot for a perfect response, when stype = 0 and MATLAB does convolution:



- When a realistic simulation of a DSP filter is plotted (stype = 1), we get several renders as follows:




12
10
8
6
4
2
0
0
0.05
0.1
0.15
0.2



When comparing the plots, they fit what we thought was true, i.e. for a realistic simulation of a DSP filter, the filter is somewhat altered although it follows the original very closely.

<u>**Any problems you ran into in the lab and how you fixed them.**</u>

During the implementation of the FIR block, there were several problems with the code itself since this was a new concept to us. However, with the help of the TA's and with some algorithmic thinking, we were able to fix these problems and complete the design as showcased below.