Communication Basics
Lap Report 3
Delay and FIR

Performed by:
Aidyn Ardabek
a.ardabek@jacobs-university.de

Instructor:
Mathias Bode
m.bode@jacobs-university.de

Date of lab: 17 January 2022

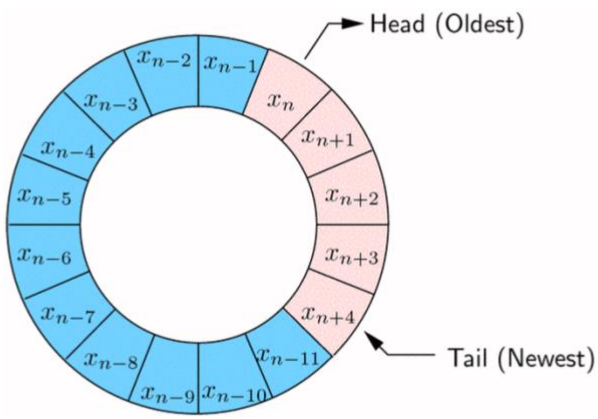
Lab Write Up:

- *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.*

First, real-time DSP codes are usually coded in C or assembly language. Also, real-time DSP code receives continuous stream of input, so it must run continuously and should give continuous output. Nevertheless, MATLAB allows us to use block diagrams in the Simulink. It means that we can reuse our code as a block with other more complex blocks to simulate real-time process.

Incoming and outgoing data packets are buffered in memory and the buffer management system ensures that there is enough memory available for the data packets. In general, poor buffer management strategy can lead to suboptimal performance. In order to process the input, the samples are divided into buffers and every of them are processed individually. Since small sized buffers are filled very fast, we have a low latency. However, we also will have a low throughput since it takes more time to store and restore many small buffers.

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



A continuous stream of samples with finite sized buffers can be simulated using circular buffers. Memory elements are arranged in a circle. A new sample is added at the tail and the first sample will always be stored at the head pointer. During the processing, a new sample enters the buffer tail, while head pointers are incremented.

When coding, we usually storage data in a line. The only problem here is that the pointers we use to access the buffer need to wrap from the end to the beginning. Usually, many DSP have built-in support for circular addressing, but in our case it requires some additional code.

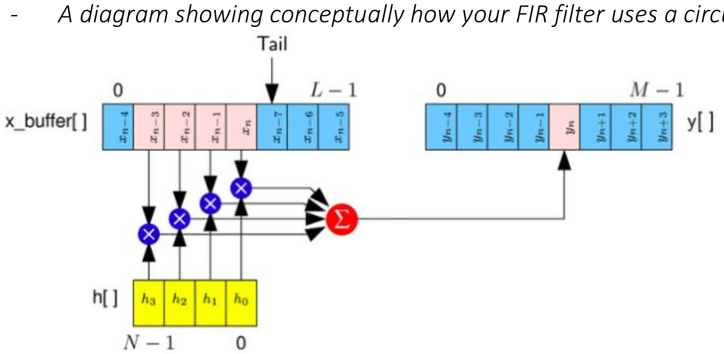
- *A printout of the MATLAB code that implements your Delay with comments.*

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

% Initializes a delay block.

% Inputs:
%   Nmax    Maximum delay supported by this block
%    N      Initial delay

% Output:
%   state   State of block

% Notes:
% For this block to operate corrently,
% you should not pass in more than Nmax
% samples at a time.

% 1. Save parameters
state.Nmax = Nmax;

% Store initial desired delay
state.N = N;

% 2. Create state variables
% 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 = state.N;
```

```
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;
```

```
% 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
Nd2 = 20;        % Delay of the second block

% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);
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
yb = zeros(Ns, Nb);
yb2 = zeros(Ns, Nb);

% Process each buffer
for bi=1:Nb
    [state_delay1, yb(:,bi)] =
               delay(state_delay1, xb(:,bi));
    [state_delay2, yb2(:,bi)] =
               delay(state_delay2, yb(:,bi));
end

% Convert individual buffers 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+Nd+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 - Nd - Nd2) ~= y(n_chk))
    warning('A mismatch was encountered.');
end
```

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



First, a buffer of N samples is received. These samples are moved into the circular buffer. By filtering them we generate N output samples. Finally, output N samples are copied into the output buffer.

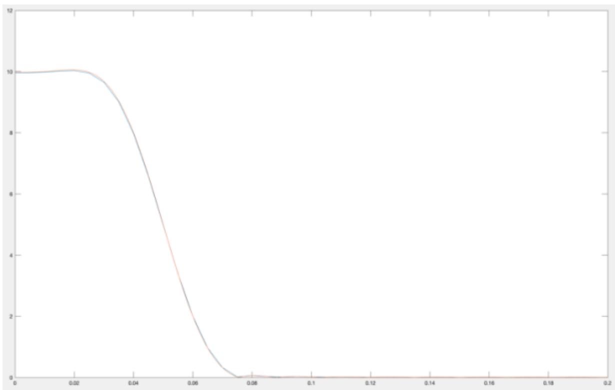
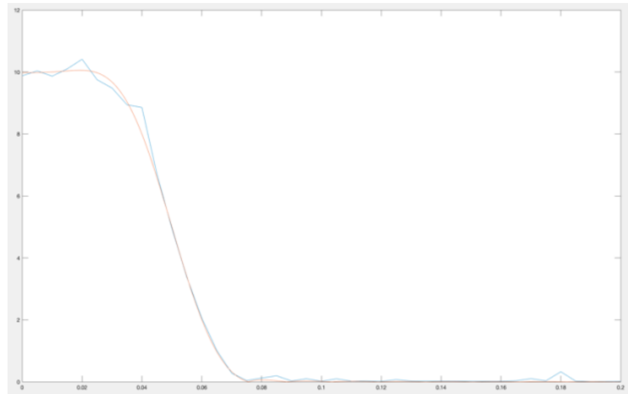
- A printout of the MATLAB code that implements your FIR filter with comments.

```
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: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);
    s.n_p=bitand(s.n_t+s.Mmask, s.Mmask);
    sum=0.0;
    for j=0:length(s.h)-1
        sum=sum+s.buff(s.n_p+1)*s.h(j+1);
        s.n_p=bitand(s.n_p+s.Mmask, s.Mmask);
    end
    y(ii+1)=sum;
end

% Return updated state
state_out = s;
```

```
function [state] = FIR_init(h, Ns)

% Creates a new FIR filter.

% Inputs:
%      h      Filter taps
%     Ns      Number of samples procesed

% Outputs:
%    state    Initial state

% 1. Save parameters
state.h = h;
state.Ns=Ns;

% 2. Create state variables
% 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(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
state.n_t=0;
state.n_p=0;
```

```
% test_fir1.m

% 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 = 0; % Do simple convolution
stype = 1;  % DSP-like filter

if stype==0
    y = conv(x, h);
elseif stype==1
  % Simulate realistic DSP filter
  state_fir1=FIR_init(h,Ns);

    % Reshape into buffers
    xb=reshape(x, Ns, Nb);

    %Output samples
    yb=zeros(Ns, Nb);

    % Process buffers
    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.

In the first plot, we used a direct convolution as in normal MATLAB processing. Whereas the second plot shows a real time simulation. The main idea is to make the response of the filter as short as possible. However, changing the efficiency of the filter affects accuracy of the filter. Having a small length makes DSP code run faster. But the accuracy of the filter becomes lower, as shown in the second plot.

- Any problems you ran into in the lab and how you fixed them.

I had some problems with indexing matrices. However, they were solved by running a program until that point, determining what caused the problem, and fixing them.