Real-Time Block Simulation

DSP and Communications Lab

2/26/2013

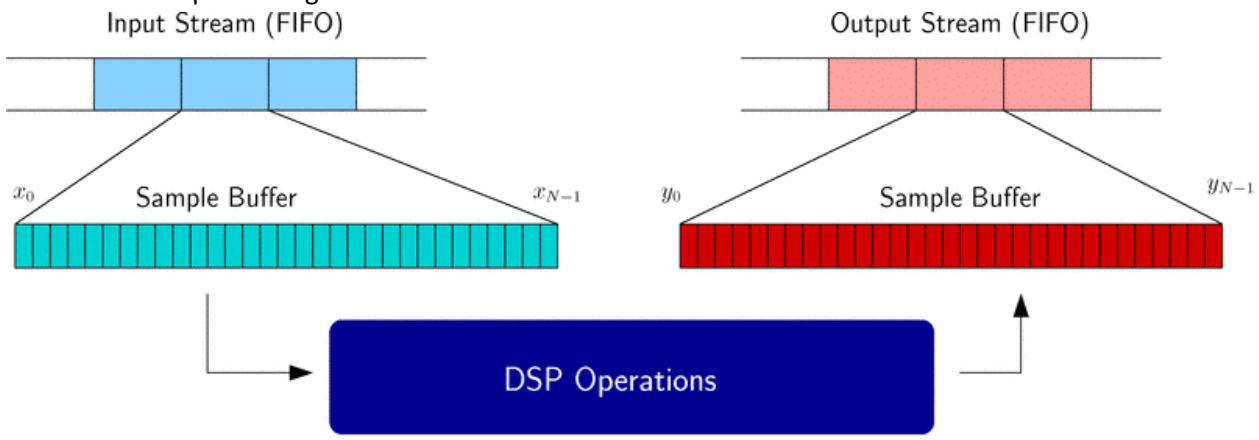
Jacobs University Bremen
Grishma Raj Pandeya and Joshan Chaudhary

1.

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.

Usually when we run a matlab simulation, the input samples that we need to process are already available. Thus, the run time (time needed by algorithm to finish the task) is not critical at all. For example, when we need to enhance a jpeg image that is already available in our hard-disk, the run time is not critical at all. Whereas, for real time DSP code, all the samples that we need to process are not available at once. Rather, we receive a continuous stream of data which needs continuous processing. An example of real time signal processing application is echo cancellation in a telephone network. Unlike matlab simulation timing is very crucial here.

Buffer oriented processing:



.

1.4 What affects latency and throughput of an algorithm?

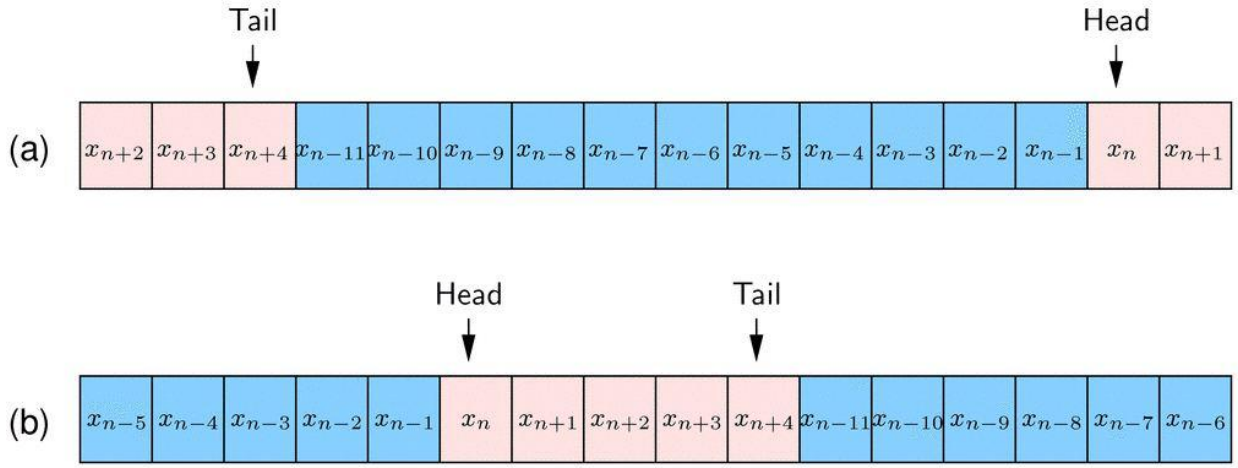
The latency and throughput of an algorithm is affected by the buffer oriented style of processing of the data. Since, we don't have all the information at once and rather we have a stream of data flowing in. we wish to create a buffer of the samples for the processing of the samples. We divide input samples to N samples to form one buffer. The size of "N" here greatly affects the latency and throughput of the algorithm. Should we choose a large N there will be a large latency between input and output of our system, Should we choose a small N the latency of the system won't be that large but it will for sure affect the throughput of the system because we will be spending a lot of time putting our samples into

the buffer before processing and to the output buffer after we are done processing with them. So, at the end it is always a tradeoff between latency and throughput. Basically, the choice of a system that we wish to design determines the size of N.

2)

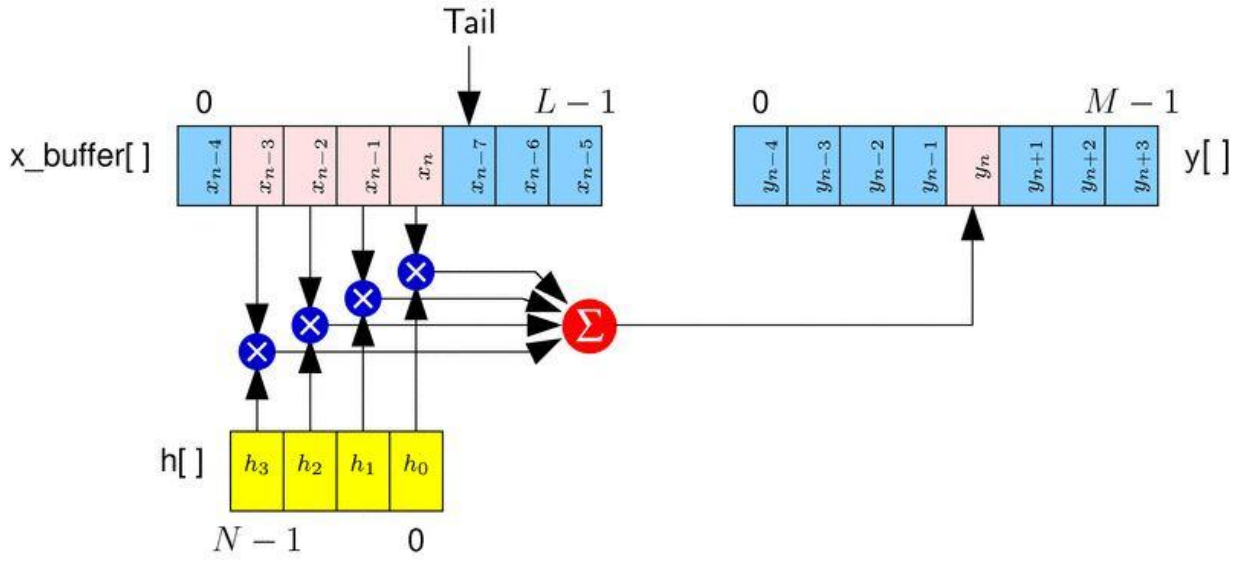
A diagram showing conceptually how your FIR filter uses a circular buffer to implement the FIR equation

First of all, the message is divided into buffers. Then we move the buffer into the circular buffer for processing one by one. To achieve a circular buffer, we use two pointers, a head pointer and a tail pointer, so that we know where to begin the processing from. Once we have all the samples in buffer nicely moved to our apparently circular buffer, we implement the filter equation on the elements of the buffer.



Above in the picture we can see the use of tail pointer to achieve circular buffer arrangements and the use of head pointer to keep track of the first element in the buffer.

Now we create a nested loop where the outer loop increments the head pointer and the inner loop implements the FIR filter equation. Doing this for each buffer, finally we get the output.



So, this is how we conceptually implement the FIR filter.

3)

A printout of the MATLAB code that implements your filter.

grp_fir_init.m

```
function [state] = grp_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

%% 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.L = 2^(ceil(log2(Ns+length(state.h))+1));
%% Temporary storage for circular buffer
state.Lmask = state.L-1;
state.buff = zeros(state.L, 1);

%% Set initial tail pointer and temp pointer (see pseudocode)
state.n_t = 0;
state.n_p = 0;
```

grp_fir.m

```
function [state_out, y] = grp_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 i=0: length(x)-1,
   s.buff(s.n_t+1) = x(i+1);
   s.n_t = bitand(s.n_t+1, s.Lmask);
% Filter samples and move into output

s.n_p = bitand(s.n_t + s.Lmask, s.Lmask);
sum = 0.0;
for l=0: length(s.h)-1,
   sum = sum + s.buff(s.n_p+1)* s.h(l+1);
   s.n_p = bitand(s.n_p + s.Lmask, s.Lmask);
end
y(i+1) = sum;
end


% Return updated state
state_out = s;
```

```
% grp_test_fir1.m
%
% Script to test the FIR filter.
```

```
% Global parameters
Nb = 100;          % Number of buffers
Ns = 32;% Samples in each buffer

% Generate filter coefficients
p.beta = 0.5;
p.fs = 0.1;
p.root = 0;        % 0=rc 1=root rc
M = 128;
[h f H Hi] = win_method('rc_filt', p, 0.2, 1, M, 0);

% we need to initialise the fir block
state_fir1 = grp_fir_init(h, Ns);



% 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
x = reshape(x, Ns, Nb);
y = zeros(Ns, Nb);
for bi = 1:Nb,
    [state_fir1, y(:,bi)] = grp_fir(state_fir1, x(:,bi));
end
  % ADD YOUR CODE HERE !!!
y = reshape(y, Ns*Nb, 1);
%x = reshape(x, Ns*Nb, 1);
else
  error('Invalid simulation type.');
end

% Compute approximate transfer function using PSD
Npsd = 200;       % Blocksize (# of freq) for PSD
[Y1 f1] = psd1(y, Npsd);
```

```
[X1 f1] = psd1(x, Npsd);
plot(f1, abs(sqrt(Y1./X1)), f, abs(H), 'r');
xlim([0 0.15]);
```

4)

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

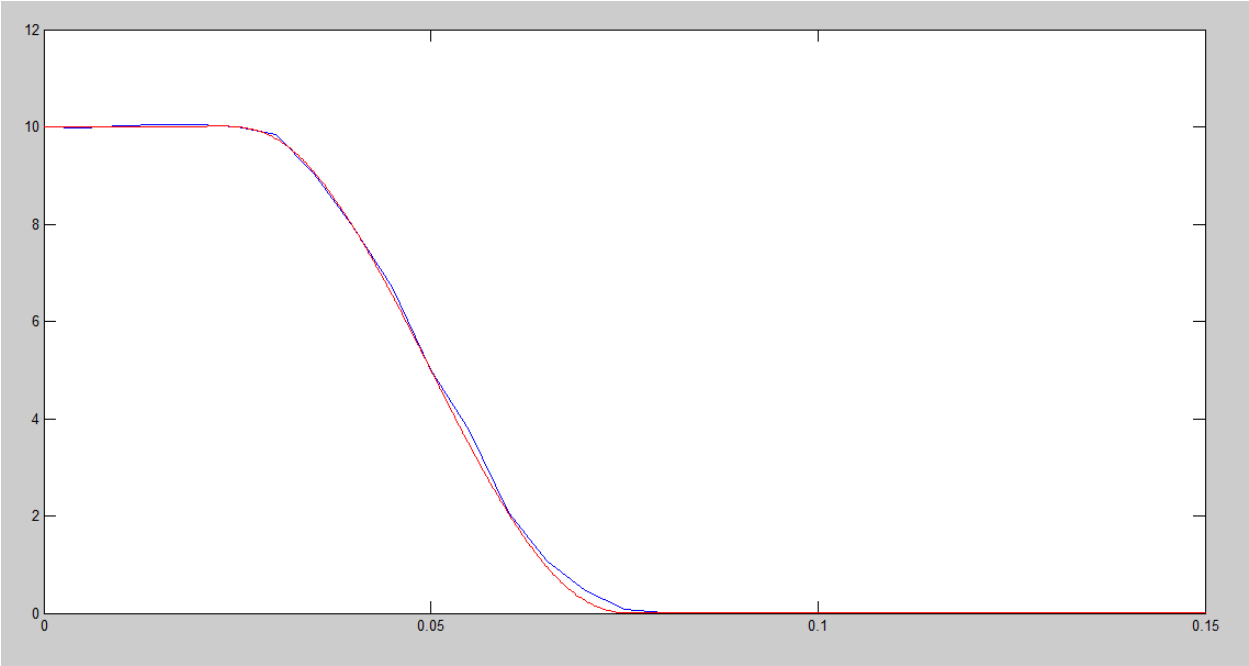


Fig:1 plot showing the ideal response and the simulated response

The red one is the ideal response and the blue one is the simulated response.

Observing the above plots we can conclude that our filter's response is quite similar to that of the ideal filter. However, there are some discrepancies which due to the fact that we break our input signal into buffers and we process them one after the other so our filter processing is not continuous.

5)

Any Problems that you ran into in the lab and how you fixed them.

We had a little problem understanding the codes in the beginning. But comparing it with the diagrams and the pseudo codes made it a lot easier to understand.

We took some wrong procedures and values in the cascaded delay blocks because of which we were getting the output delayed by two times more than desired. Later, by consulting with a TA and revising the code we solved it.

Lastly, we made a mistake implementing the nested for loop because of which the code was not compiling. It was again solved by consulting with a TA and revising the code.