Michael Carroll – mjcarroll@ieee.org

ELEC6410 –Digital Signal Processing

30 August, 2010

# Project 1

## Exercise 1

The purpose of Exercise 1 was to generate a periodic, truncated, decaying exponential function.

### Code Listing

function x = Exercise1( L, A, b, M )

%% Exercise1

% Generate a periodic, truncated, decaying exponential function:

% L - Total number of samples in the waveform

% A - Beginning amplitude of the exponential function

% b - Decay rate of the exponential function

% M - Number of samples each period will last

% The function takes the form $A\*\exp{-bn}$

x = A\*exp(-b \* mod([0:L-1],M));

end

### Example Generated Output

To test the function, a sample input of L = 80, A = 2, b = 0.08, and M = 20 was used. The output is included as Figure 1. Four decaying exponential functions are shown, as to be expected from the given inputs.

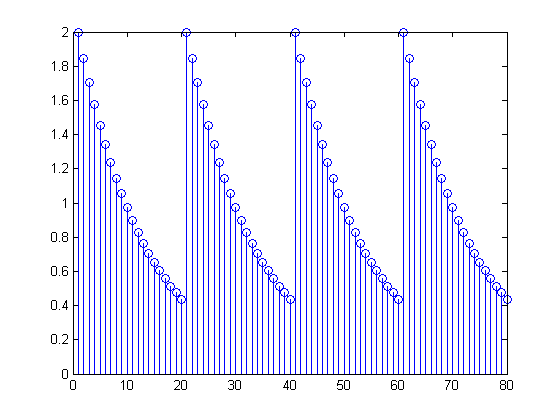


Figure : Sample Output of Exercise1.m Function

## Exercise 2

The purpose of Exercise 2 was to create two signals for convolution. The code for generating the two signals is included in Appendix A – Exercise2th5.m, Exercise 2.

## Exercise 3

Exercise 3 builds on the previous exercise by convolving the two signals generated in Exercise 2. The code for the convolution is included in Appendix A – Exercise2th5.m, Exercise 3. The output of the exercise is given in Figure 2.

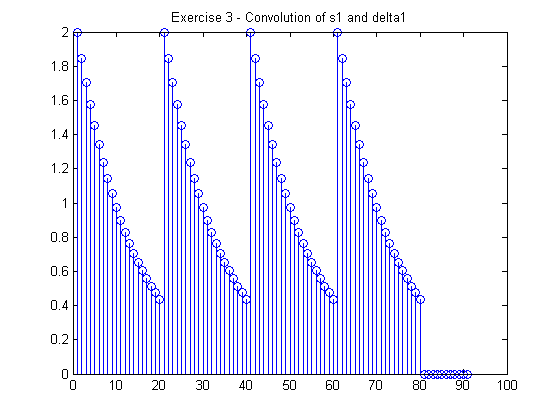


Figure : Output of conv(s1,delta1)

## Exercise 4

Exercise 4 builds once again on the previous exercise, but changes the delta function used for convolution. The code for the convolution is included in When compared to the output of the previous convolution function in Figure 2, it can be seen that the new convolution is a sum of the two periodic, exponential waveforms. This can be explained by the addition of the second impulse at array index 12. On the left end of the waveform, the samples are all multiplied by 2. On the right end of the waveform, only the effects of the second impulse are seen in the convolution. In the middle, a variety of periodic samples appear from the function of convolution delta2 having the additional impulse.

## Exercise 5

The purpose of Exercise 5 was to generate three normalized waveforms to be used in the conv function. The code used to generate the waveforms and plots is included in **Error! Not a valid bookmark self-reference.**, Exercise 5. The output stem plots are included as Figure 4, Figure 5, and Figure 6

Figure 4: Output of conv(hn3,s1) Figure 5: Output of conv(hn5,s1)

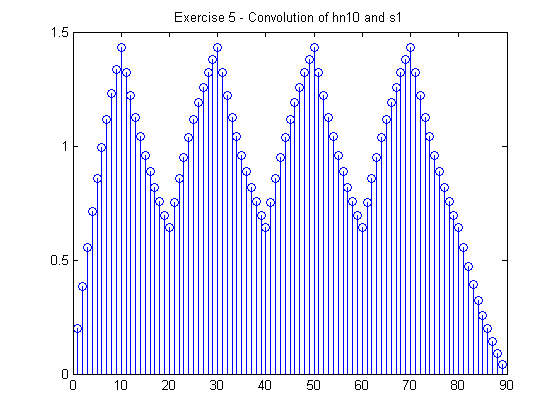


Figure 6: Output of conv(hn10,s1)

Comparing the three plots, there are a few major differences. The first difference is in the peak magnitude of the three plots. The unit step functions that are normalized for length weight each function differently, causing the amplitude of the outputs to be inversely proportional to the length of the convolved step function.

The second primary difference is the output of conv(hn3,s1) is overall closer to the original input function hn3, while the output of conv(hn10,s1) more closely represents the hn10 function. This would seem to imply that the signal with more samples as the input to the convolution function will manifest itself stronger in the output.

## Exercise 6

Start with the definition of a convolution:

Given the two discrete-time functions:

Plugging them into the convolution formula:

The inside of the summation can be expanded (since this is a linear-shift invariant system):

Which can reduce to:

This expressed as a vector would be:

[1 1 1 3 3 2 2 2]

To check this, two vectors must be created in MATLAB, one for x[n] and one for y[n]:

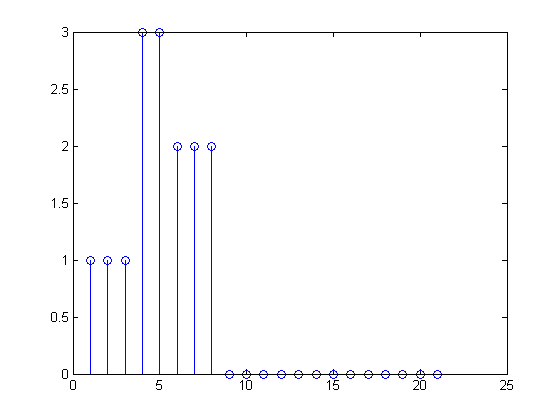
x = [1 1 1 1 1 0 0 0 0 0 0];

h = [1 0 0 2 0 0 0 0 0 0 0];

out = conv(x,h)

stem(out)

out = [1 1 1 3 3 2 2 2 0 0 0];



## Exercise 7 (6410)

The purpose of Exercise was to evaluate two iterative squre root functions, and compare their performance over a range of values between 0 and 1. The first function is included as Appendix B.1 - Exercise7a.m, and the second function is included as Appendix B.2 – Exercise7b.m. The two functions are compared in a separate benchmarking script, which is included as Appendix B.3 - Exercise7.m.

The first step of the comparison was to visualize the two equations converging on the desired value. This is included below as Figure 7.

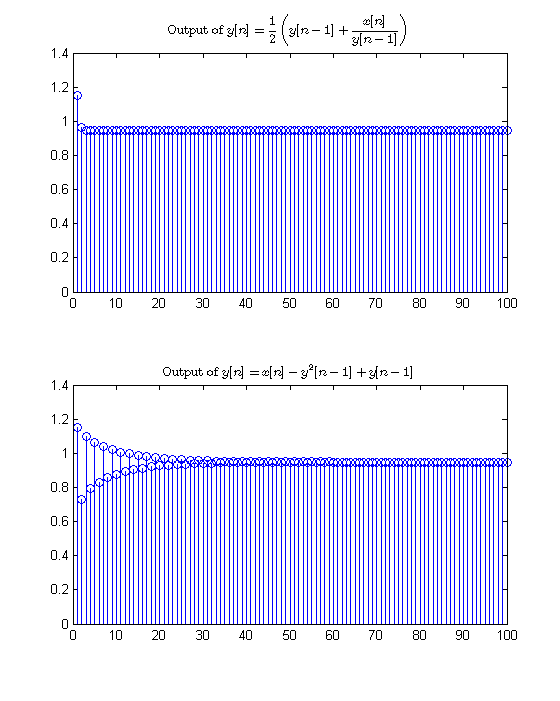


Figure 7: Convergence of Two Square Root Functions

The two functions were then evaluated with an array of alpha values, 1000 evenly spaced values between 0 and 1, inclusive. The first function was significantly faster, with mean iterations being 3.6 versus 13.35 for the second function. This is almost an order of magnitude faster. The difference becomes most evident at values close to 0 and 1.

Appendix A – Exercise2th5.m, Exercise 4. The output of the exercise is given in Figure 3.

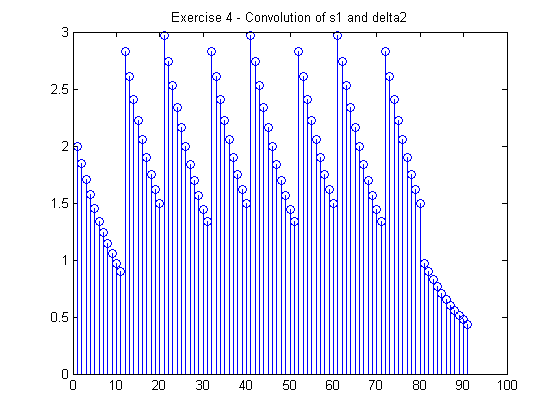


Figure : Output of conv(s1,delta2)

When compared to the output of the previous convolution function in Figure 2, it can be seen that the new convolution is a sum of the two periodic, exponential waveforms. This can be explained by the addition of the second impulse at array index 12. On the left end of the waveform, the samples are all multiplied by 2. On the right end of the waveform, only the effects of the second impulse are seen in the convolution. In the middle, a variety of periodic samples appear from the function of convolution delta2 having the additional impulse.

## Exercise 5

The purpose of Exercise 5 was to generate three normalized waveforms to be used in the conv function. The code used to generate the waveforms and plots is included in **Error! Not a valid bookmark self-reference.**, Exercise 5. The output stem plots are included as Figure 4, Figure 5, and Figure 6

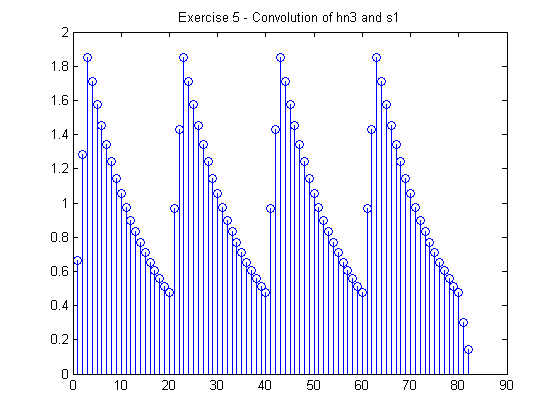
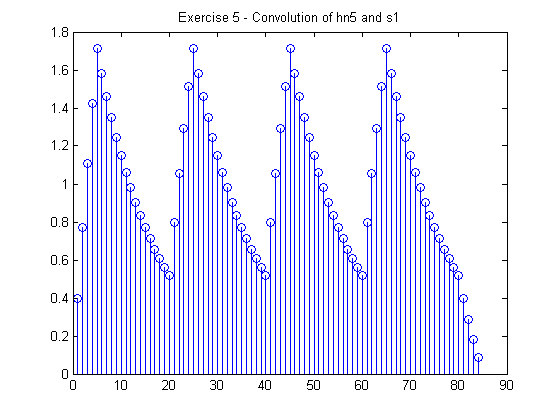


Figure : Output of conv(hn3,s1) Figure : Output of conv(hn5,s1)

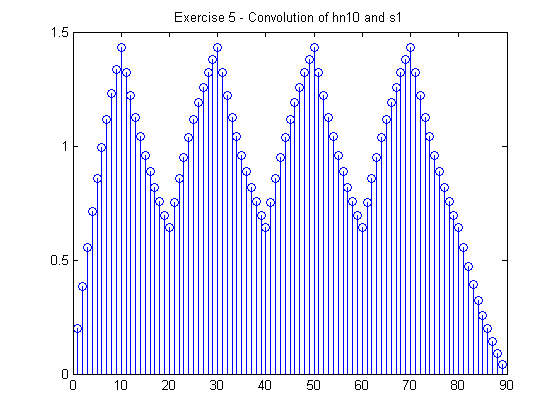


Figure : Output of conv(hn10,s1)

Comparing the three plots, there are a few major differences. The first difference is in the peak magnitude of the three plots. The unit step functions that are normalized for length weight each function differently, causing the amplitude of the outputs to be inversely proportional to the length of the convolved step function.

The second primary difference is the output of conv(hn3,s1) is overall closer to the original input function hn3, while the output of conv(hn10,s1) more closely represents the hn10 function. This would seem to imply that the signal with more samples as the input to the convolution function will manifest itself stronger in the output.

## Exercise 6

Start with the definition of a convolution:

Given the two discrete-time functions:

Plugging them into the convolution formula:

The inside of the summation can be expanded (since this is a linear-shift invariant system):

Which can reduce to:

This expressed as a vector would be:

[1 1 1 3 3 2 2 2]

To check this, two vectors must be created in MATLAB, one for x[n] and one for y[n]:

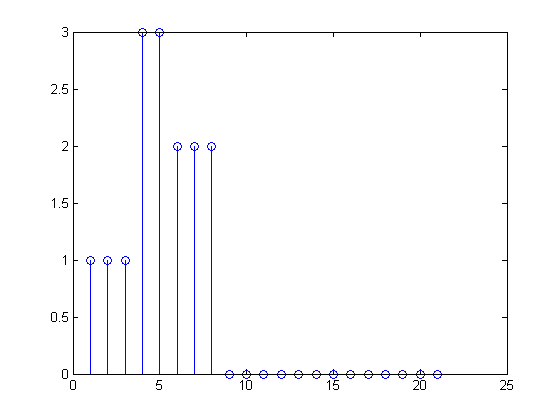
x = [1 1 1 1 1 0 0 0 0 0 0];

h = [1 0 0 2 0 0 0 0 0 0 0];

out = conv(x,h)

stem(out)

out = [1 1 1 3 3 2 2 2 0 0 0];



## Exercise 7 (6410)

The purpose of Exercise was to evaluate two iterative squre root functions, and compare their performance over a range of values between 0 and 1. The first function is included as Appendix B.1 - Exercise7a.m, and the second function is included as Appendix B.2 – Exercise7b.m. The two functions are compared in a separate benchmarking script, which is included as Appendix B.3 - Exercise7.m.

The first step of the comparison was to visualize the two equations converging on the desired value. This is included below as Figure 7.

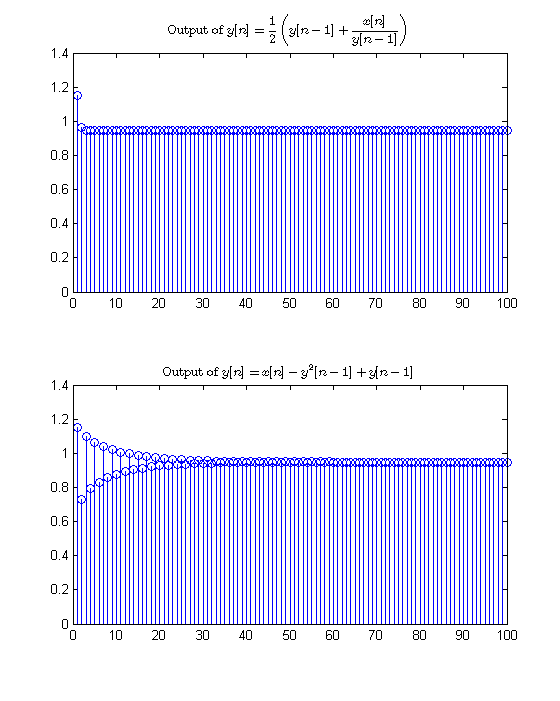


Figure : Convergence of Two Square Root Functions

The two functions were then evaluated with an array of alpha values, 1000 evenly spaced values between 0 and 1, inclusive. The first function was significantly faster, with mean iterations being 3.6 versus 13.35 for the second function. This is almost an order of magnitude faster. The difference becomes most evident at values close to 0 and 1.

# Appendix A – Exercise2th5.m

%% Exercise 2

% Create two signals:

% (a) Create a length-12 vector representing an impulse 0 and 11.

delta1 = [1 zeros(1,11)];

% (b) Create a wave s1 with L = 80, A = 2, b = 0.08, and M = 20

s1 = Exercise1(80,2,0.08,20);

%% Exercise 3

% Use conv to convolve s1 and delta1, and plot the result using stem.

exercise3 = figure;

stem(conv(s1,delta1))

title('Exercise 3 - Convolution of s1 and delta1')

%% Exercise 4

% Examine another convolution:

% (a) Create another vector of length 12 representing an impulse at 0 and 11.

delta2 = [1 zeros(1,10) 1];

% (b) Convolve delta2 with s1 and plot the result

exercise4 = figure;

stem(conv(s1,delta2))

title('Exercise 4 - Convolution of s1 and delta2')

%% Exercise 5

% Examine another type of impulse response:

% (a)Create a flat impulse response hn3 that is three points long and

% normalized by the length

hn3 = 1/3 \* [ones(1,3)];

% (b) Convolve hn3 with s1

exercise5a = figure;

stem(conv(s1,hn3))

title('Exercise 5 - Convolution of hn3 and s1')

% (c) Increase the length of the impulse response to 5 and 10 and redo the

% convolution

hn5 = 1/5 \* [ones(1,5)];

hn10 = 1/10 \* [ones(1,10)];

exercise5c1 = figure;

stem(conv(s1,hn5))

title('Exercise 5 - Convolution of hn5 and s1')

exercise5c2 = figure;

stem(conv(s1,hn10))

title('Exercise 5 - Convolution of hn10 and s1')

# Appendix B - Exercise7

## Appendix B.1 - Exercise7a.m

function y = Exercise7a(x, yInit)

%% Exercise7a

% Compute an iterative square root from a given input:

% $$ x[n] = \alpha\*\mu[n] $$

% Using the formula:

% $$ y[n] = 1/2 \* (y[n-1] + \frac{x[n]}{y[n-1]})$$

iterations = size(x,2);

% Pre-allocate, because it seems to make MATLAB happy.

y = [zeros(1,iterations)];

% Handle the first case here

y(1) = 1/2 \* (yInit + x(1)/yInit);

% Take care of the rest of our iterations

for iter = 2:iterations

y(iter) = 1/2 \* (y(iter-1) + x(iter)/y(iter-1));

end

end

## Appendix B.2 – Exercise7b.m

function y = Exercise7b(x, yInit)

%% Exercise7b

% Compute an iterative square root from a given input

% $$ x[n] = \alpha\*\mu[n] $$

% Using the formula:

% $$ y[n] = x[n] - y^{2}[n-1] + y[n-1]$$

iterations = size(x,2);

% Pre-allocate, because it seems to make MATLAB happy.

y = [zeros(1,iterations)];

% Handle the first case here

y(1) = x(1) - yInit^2 + yInit;

% Take care of the rest of our iterations

for iter = 2:iterations

y(iter) = x(iter) - y(iter-1)^2 + y(iter-1);

end

end

## Appendix B.3 - Exercise7.m

%% Exercise 7

% Compare two different iterative square-root functions.

%

% Given an input $x[n] = \alpha\mu[n]$, where $\alpha$ is the number whose

% square root is desired. For this system $\alpha$ must be between 0 and

% 1.

%% User inputs

% Provide a seed value y[-1], for the calculations

yInit = 0.5;

% Provide $\alpha$, the number whose square root we are calculating

alpha = .9;

% Provide the number of iterations to calculate

iterations = 100;

%% Generate Working Values

x = alpha \* ones(1,iterations);

%% Calculate the square roots

Output7a = Exercise7a(x,yInit);

Output7b = Exercise7b(x,yInit);

%% Plot

OutputFig = figure;

subplot(2,1,1);

stem(Output7a)

title('Output of $\displaystyle y[n] = \frac{1}{2} \left(y[n-1] + \frac{x[n]}{y[n-1]}\right)$',...

'interpreter','latex')

subplot(2,1,2);

stem(Output7b)

title('Output of $\displaystyle y[n] = x[n] - y^2[n-1] + y[n-1]$', ...

'interpreter','latex')

%% Efficiency Comparison

% Evaluate how many iterations each function takes to get within 4 decimal

% places of accuracy over a range of inputs.

% Check 100 input values between 0 and 1, for statistics purposes.

inputs = linspace(0,1,1000);

outputs = zeros(2,size(inputs,2));

% I'm going to go loop-heavy here because it's non-critical code

for iter = 1:size(inputs,2)

x = inputs(iter) \* ones(1,iterations);

compare = sqrt(x(1));

Output7a = Exercise7a(x,yInit);

Output7b = Exercise7b(x,yInit);

for n = 1:size(Output7a,2)

E = abs(compare - Output7a(n));

if E < 0.0001

outputs(1,iter) = n;

break

end

end

for n = 1:size(Output7a,2)

E = abs(compare - Output7b(n));

if E < 0.0001

outputs(2,iter) = n;

break

end

end

end

% Display the average of all of the iterations required over a range of

% input values.

averages = mean(outputs,2)