Wavelets And Applications Coursework: Sampling Signals with Finite Rate of Innovation with an Application to Image Super Resolution

Jacobus Hertzog
jjh113@ic.ac.uk

27th March 2017

Contents

1 Strang-Fix Conditions

1.1 Exercise 1

In this exercise, we show that a function satisfying the Strang-Fix conditions with N+1 moments is able to reproduce polynomials of any order up to N. The Strang-Fix conditions are important as any function satisfying them is a valid member of the polynomial reproducing kernel family. Daubechies scaling functions are an example of functions that satisfy the Strang-Fix conditions and are thus members of this family. As a result if we wish to reproduce polynomials of order 0-3 with a Daubechies scaling function then we require a Daubechies filter with $N+1=4$ vanishing moments. Therefore we use the scaling function named 'dB4', shown in figure 1.

Initially $32-L$ coefficients were used, as directed in the coursework specification. L is the support, the range of values for which the scaling function is non-zero. For Daubechies wavelets, the support width is given by $2N-1$, so for 'dB4', the support is 7. This produced 26 coefficients.However, it was discovered that the error of the reconstruction could be reduced when using 32 coefficients. Therefore, 32 coefficients were calculated and used for the remainder of the project.

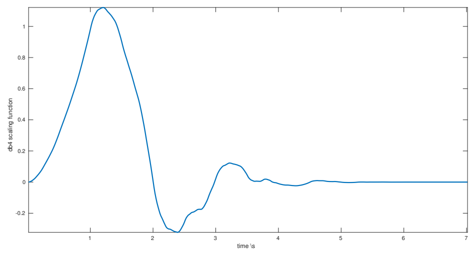

Figure 1: dB4 Scaling Function

As Daubechies wavelets are orthogonal, so there is no need to calculate the dual basis of the signal in order to find the reconstruction coefficients. The coefficients were instead given by the formula $c_{m,n} =< t^m, \varphi(t/T-n) >$. Once the coefficients are computed, and divided by 64 to account for the sampling rate, they can be multiplied with the appropriate shifted sampling kernel in order to reproduce any piecewise polynomial up to the order given by N.

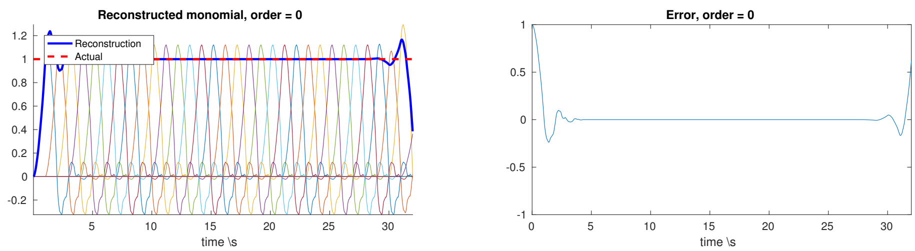

Figure 2: Reconstruction of 0 order monomial

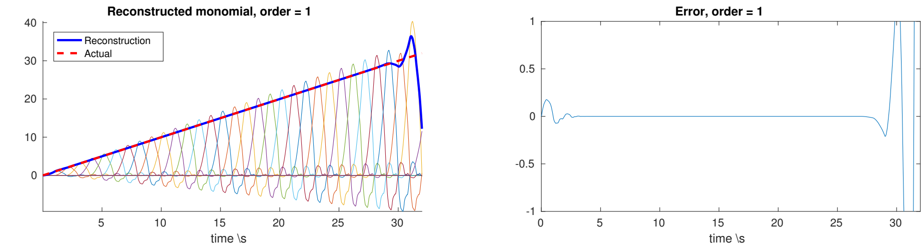

Figure 3: Reconstruction of 1 order monomial

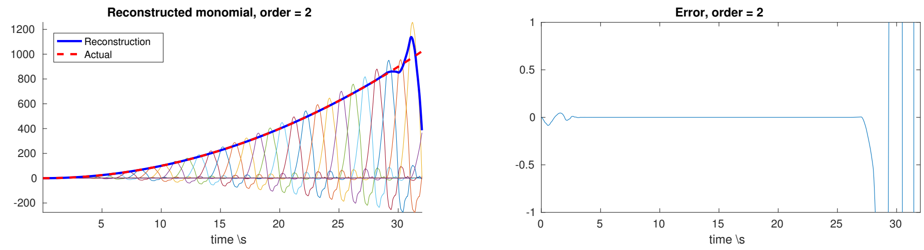

Figure 4: Reconstruction of 2 order monomial

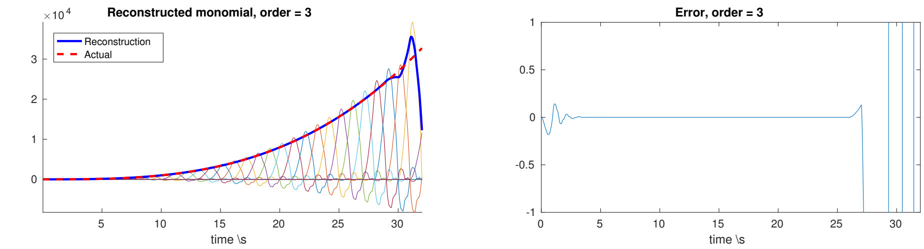

Figure 5: Reconstruction of 3 order monomial

The plots in figures 2,3, 4, and 5 show that the sampling kernel is able to reproduce the polynomials perfectly, excluding the edges of the reconstruction where the reconstruction has no preceding/-following samples to carry on the reconstruction. However, when we attempt to reconstruct a polynomial of order 4, as shown in figure 6, errors are introduced.

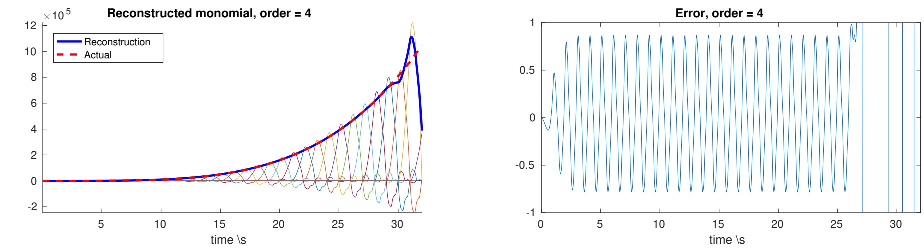

Figure 6: Attempted reconstruction of 4 order monomial

1.2 Exercise 2

The next exercise was similar, but a bit more complex. Rather than using a Daubechies scaling function, a B-Spline was used. B-Splines are produced by iteratively convolving a rectangular function, or Haar wavelet. In order to reproduce polynomials of order 0-3, a B-Spline of order 3, β_3, was produced by convolving the Haar wavelet four times. The result is shown in figure 7

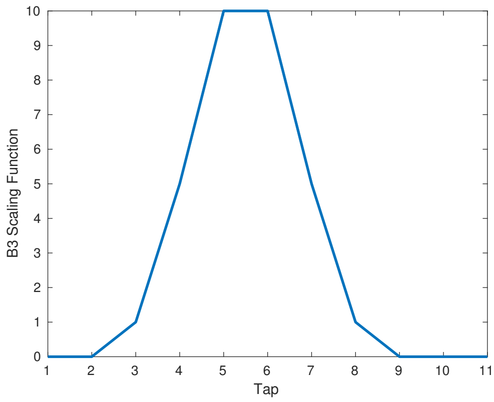

Figure 7: β_3 Scaling Function

With the exception of β_0, which is equivalent to the Haar wavelet, B-Splines are not orthogonal, and as a result, it is necessary to calculate the dual basis of β_3 in order to find the coefficients for the polynomial reproduction. Unfortunately, I was not able to compute the dual basis of β_3 and was therefore unable to complete the rest of the exercise.

2 The Annihilating Filter Method

2.1 Exercise 3

This exercise involved writing a program that would find the annihilating filter of a signal $\tau[m]$. This was a straightforward case of following the instructions given. Given that $h[1] = 1$, the annihilating filter was found by solving equation 2.1:

$$\begin{pmatrix} h[1] \\ h[2] \\ \vdots \\ h[K] \end{pmatrix} = \begin{bmatrix} \tau[K-1] & \tau[K-2] & \cdots & \tau[0] \\ \tau[K-1] & \tau[K-1] & \cdots & \tau[1] \\ \vdots & \vdots & \ddots & \vdots \\ \tau[N-1] & \tau[N-2] & \cdots & \tau[N-K] \end{bmatrix}^{-1} - \begin{pmatrix} \tau[K] \\ \tau[K+1] \\ \vdots \\ \tau[N] \end{pmatrix} \tag{1}$$

Once the annihilating filter was found, the locations of the Diracs, t_k, could be easily found by finding the roots of h. Next, the Vandermonde system in equation 2.1 was solved to find the weights, a_k.

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{K-1} \end{pmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ t_0 & t_1 & \cdots & t_{K-1} \\ \vdots & \vdots & \ddots & \vdots \\ t_0^{K-1} & t_1^{K-1} & \cdots & t_{K-1}^{K-1} \end{bmatrix}^{-1} \begin{pmatrix} \tau[0] \\ \tau[1] \\ \vdots \\ \tau[K-1] \end{pmatrix} \tag{2}$$

This algorithm was applied to the τ variable obtained from`tau.mat`, the file provided to us. The annihilating filter $h[n]$ value are shown in table 1 and the recovered locations and weights are shown in table 2. The reconstructed stream of Diracs is shown in figure 8.

$h[0]$	$h[1]$	$h[2]$
1.0000	-29.6250	219.0937

Table 1: The annihilating filter coefficients.

k	t_k	a_k
0	15.3750	0.7800
1	14.2500	1.3200

Table 2: Results of the annihilating filter applied to `tau.mat`.

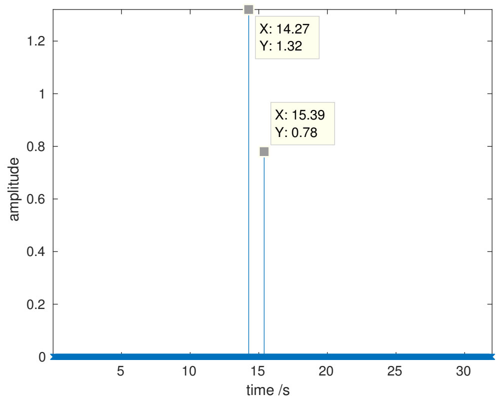

Figure 8: Reconstruction of Dirac Stream

To verify that the annihilating filter is correctly constructed, we can perform a simple test. By definition, the convolution of the annihilating filter h with the τ should result in a zero output at the centre. The result of the convolution $y = h * \tau$, is shown in 3. The only important results here are those at index 2 and 3, as the other values are cases where the convolution cannot shift the signals fully through one another. Since these are both zero

$y[0]$	$y[1]$	$y[2]$	$y[3]$	$y[4]$	$y[5]$
2.10	-31.40	0	0	-98016.19	1457963.78

Table 3: Results of the convolution $y = h * \tau$

Since this method was to be reused for other exercises, it was developed into a function called `annihilatingFilterMethod.m`

3 Sampling Diracs

3.1 Exercise 4

This exercise required us to construct our own Dirac stream, sample it with a sampling kernel, and perform the annihilating filter method to reconstruct the signal from the samples. The first step in this process was selecting parameters for the Dirac stream. The Dirac stream is given by the equation $x(t) = \sum_{k=0}^{K-1} a_k \delta(t - t_k)$, so when $K = 2$, we have a stream with two Diracs and we need to select two values for location and weight. These are given in the table 4.

k	t_k	a_k
0	12.5	5
1	23.0	2

Table 4: Original weights and locations of $x(t)$

k	t_k	a_k
0	12.5	5
1	23.0	2

Table 5: Retrieved weights and locations from the annihilating filter method

The signal was sampled using the same 'dB4' scaling function as shown in figure 1 from exercise 1. This is because we know that the sampling kernel $\varphi(t)$ must be able to reproduce polynomials of order $N \geq 2K - 1$, so for $K = 2$, $N \geq 3$.

The next step in the process was to sample the signal. The samples were produced using the formula $y_n =< x(t), \varphi(t/T - n) >$. Once these samples had been computed, the moments of the signal were calculated as the product of the samples and the sampling kernel's coefficients. Since the same scaling function is used as in exercise 1, the coefficients were simply reused. The sequence of moments was thus given by the formula $\tau(m) = \sum_n c_{m,n} y[n]$.

Now that the sequence of moments had been found, the signal was recovered using the annihilating filter function `TODO anni`, developed in exercise 3. The retrieved locations and weights are shown in 5 results of the reconstruction are shown in figure 9. It can be clearly seen that the reconstruction fully matches the original signal.

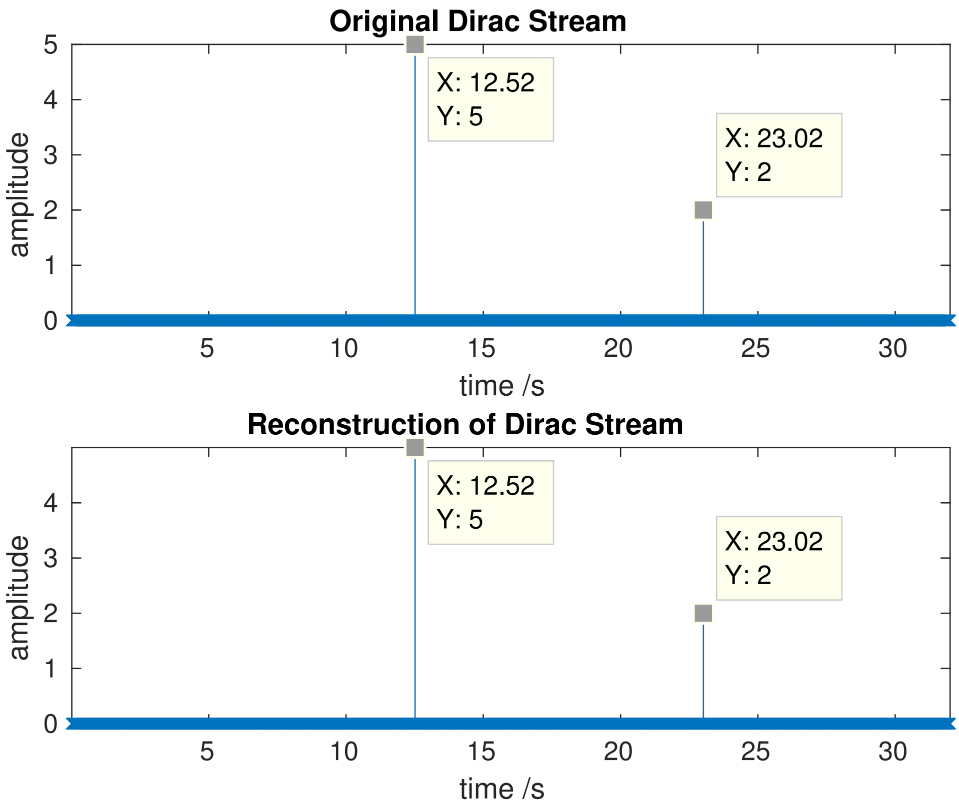

Figure 9: Original Signal and Reconstruction

3.2 Exercise 5

In this exercise, the signal samples from an unknown signal were provided in the file `sample.mat`. By applying the same method as in exercise 4, this signal could be reconstructed. As described previously, the moments were computed using the formula $\tau(m) = \sum_n c_{m,n} y[n]$, and the annihilating filter method was applied. The results of the reconstruction are shown in table 6 and figure 10.

k	t_k	a_k
0	14.3750	2.6300
1	17.5000	1.4800

Table 6: The location and weights of the Diracs from the unknown signal.

This result can be verified by taking samples of the reconstructed signal and checking whether this matches the original set of samples. Using the 'db4' scaling function and the formula $y_n =< x(t), \varphi(t/T - n) >$ from the previous exercise, the samples were calculated and are shown in 10 alongside the original samples. The two waveforms are identical, verifying that the reconstructed signal is correct.

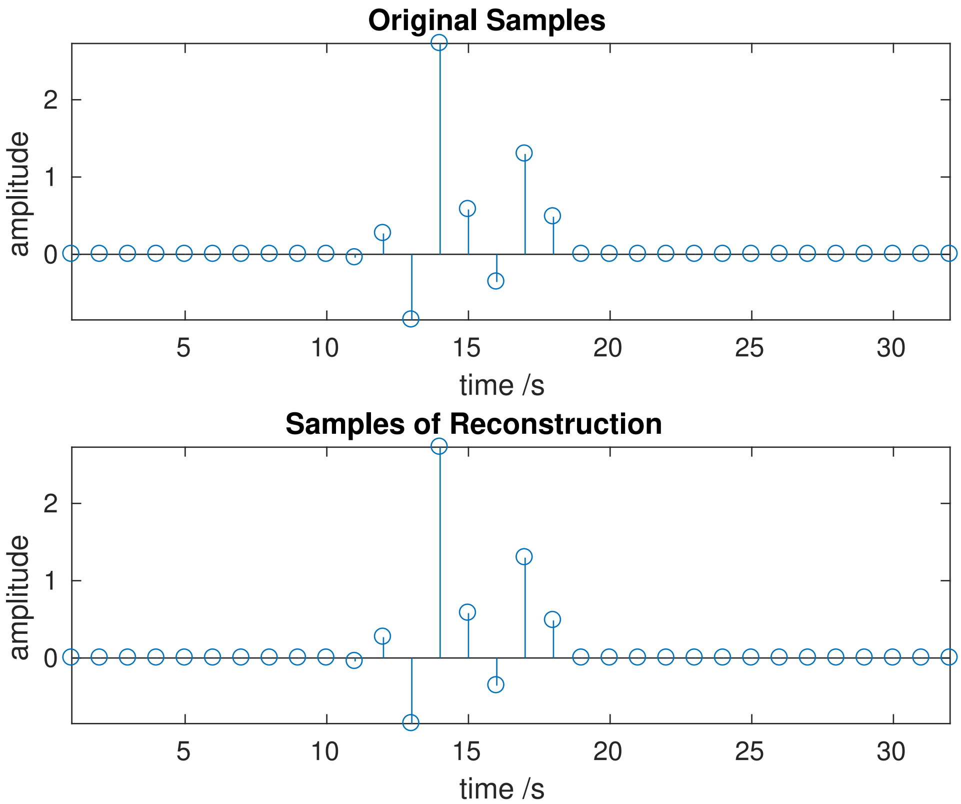

Figure 10: Original Signal and Reconstruction

4 Reconstruction in the Presence of Noise

4.1 Exercise 6

This exercise simply involved calculating generating a Dirac stream and finding the moments as previously done, and then adding Gaussian noise to the moments. This also needs to be done for different values of N and variances, σ^2 of the Gaussian noise.

In order to make this process easier, a function named `generateMoments` was written that would take in a Dirac stream and the desired N value, generate the appropriate Daubechies scaling function 'dBN', generate the polynomials t^m, obtain the coefficients, sample the signal and finally combine the coefficients and the samples to procude the moments of the signal. We are given that $N > 2K$. Note that because noise is added, $N = 2K$ is no longer viable as it has been in previous exercises. $K = 2$, so $N > 4$. The range $N = [5 : 8]$ was chosen for testing.

After each sequence of moments was generated, noise was added to it. The noise values were generated simply using the `randn` to generate normally distributed values. Multiplying these values by the standard deviation, σ, produced the noise values that were added to the moments. The range of moments selected was $\sigma^2 \in 0.001, 0.01, 0.1, 1, 10$. An example of a sampling kernel and noisy moment signal are shown in figure 11.

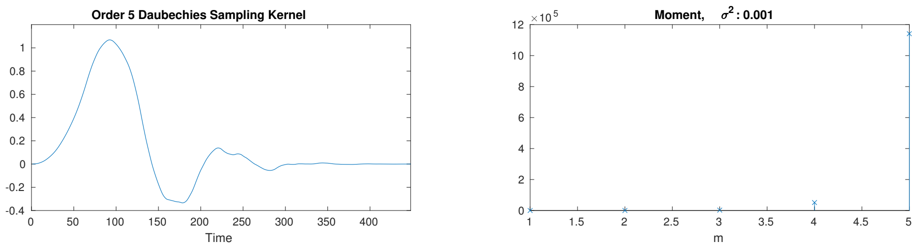

Figure 11: Sampling Kernel and Moments Signal with Noise Added

4.2 Exercise 7

In this exercise, the reconstruction of the signals from noisy moments was attempted. This could be done using the standard annihilating filter seen in previous exercises, but only with limited success. It can be improved upon using the Total Least-Squares (TLS) approach. This involves computing the annihilating filter using Singular Vector Decomposition(SVD). A Toeplitz matrix S is broken down into $S = U\Lambda V^T$. The final column of V is then taken to be the annihilating filter $h[n]$. Once this is complete, the locations t_k and the weights a_k are found in the standard way.

Beyond this, the results can be further improved by performing the Cadzow routine upon the Toeplitz matrix S. The SVD is taken again, producing $S = U\Lambda V^T$, and then the Λ matrix is modified. This is a diagonal matrix containing the eigenvalues of S, and all but the K largest eigenvalues are set to zero to produce Λ'. These values set to zero are components of the noise, so by setting them to zero, the matrix is being 'denoised'. Then S' is reformed as $S' = U\Lambda' V^T$, and the process may be repeated to reduce the noise further, or the LTS method can be performed to find estimates of the Dirac stream.

The normal annihilation method was able to detect two Diracs in the stream, but there were heavy errors present in the estimations of location and weight that grew larger as the noise increased. This is expected, as errors introduced to the values of the moments will naturally throw off the estimations. There was no observable variation with different values of N.

When TLS was introduced, it was more resistant to noise until $\sigma^2 = 10$ when the system could no longer detect two Diracs. Cadzow improved the system even further, allowing detection more accurately at higher noise levels. Unfortunately I have run out of time and am not able to include these figures in the report.

A Ex1

```
%% Setup
clc
close all
clear variables

resolution = 64; maxTime = 32;
signalLength = 64*32;

time = (1:signalLength)./resolution;

%% Compute and plot Daubechies scaling function

% want to reproduce polynomials with degree 3
% scaling function must produce wavelets with 3+1=4 vanishing moments
% dbN where N is the number of vanishing moments
% db4 is selected as it has 4 vanishing moments

phi = zeros(1,signalLength);
[phi_T,¬,¬] = wavefun('db4',6);
phi(1:length(phi_T))=phi_T;
figure
plot(time(1:length(phi_T)),phi_T,'LineWidth',2)
axis tight
xlabel('time \s')
ylabel('db4 scaling function')

%% Set Up Coefficient Computation

% initially found support to be 7, supported by .. saying it was 2N-1

m_degree = 0:4;
n_numCoefficients = 32;

t = ones(4,signalLength); % t^0 and initialise rest of matrix
t(2,:) = (0:signalLength-1)./64; % t^1
t(3,:) = t(2,:).^2; % t^2
t(4,:) = t(2,:).^3; % t^3
t(5,:) = t(2,:).^4; % t^4
% Construct the vectors of t^m, m = 0:degree-1 .
% Also generate an extra t^m+1 to show failure
t = ones(m_degree(end),signalLength);
for order = m_degree+1
    t(order,:) = ((0:signalLength-1)./resolution).^(order - 1);
end

%% Compute coefficients

c_coefficients = zeros(length(m_degree),n_numCoefficients);
for mIndex = m_degree+1
    for nIndex = 1:n_numCoefficients
        shift = (nIndex-1) * resolution; % find the shift
        phiShifted = zeros(1,signalLength); % initialise phiShifted
        phiShifted((1 + shift):(length(phi_T) + shift)) = phi_T; % compute phiShifted
        phiShifted = phiShifted(1:signalLength); % crop to signalLength

```

```
        c_coefficients(mIndex, nIndex) = dot(t(mIndex,:),phiShifted)./resolution; ...
            % inner product to find c
    end
end

%% Reproduce the polynomials



t_reproduced = zeros(length(m_degree),signalLength);
error = t_reproduced;
for mIndex = m_degree+1
    title1 = sprintf('Reconstructed monomial, order = %i',mIndex-1);
    title2 = sprintf('Error, order = %i',mIndex-1);

    figure
    subplot(1,2,1)
    hold on

    % compute and plot reconstruction
    for nIndex = 1:n_numCoefficients
        shift = (nIndex-1) * resolution; % find the shift
        phiShifted = zeros(1,signalLength); % initialise phiShifted
        phiShifted((1 + shift):(length(phi_T) + shift)) = phi_T; % compute phiShifted
        phiShifted = phiShifted(1:signalLength); % crop to signalLength

        reproduction_contribution = c_coefficients(mIndex,nIndex) * phiShifted; % ...
            compute contribution at this m and n
        t_reproduced(mIndex,:) = t_reproduced(mIndex,:) + ...
            reproduction_contribution; % add to sum
        plot(time,reproduction_contribution)
    end
    h1 = plot(time,t_reproduced(mIndex,:),'b','LineWidth',2);
    h2 = plot(time,t(mIndex,:),'r--','LineWidth',2);
    legend([h1 h2],{'Reconstruction', 'Actual'}, 'Location', 'northwest')
    xlim([0 signalLength])
    xlabel('time \s')
    axis tight
    title(title1)

    % compute and plot error
    error(mIndex,:) = t(mIndex,:) - t_reproduced(mIndex,:);

    subplot(1,2,2)
    plot(time,error(mIndex,:))
    xlabel('time \s')
    axis([0 32 -1 1])
    title(title2)
end

save('reproductionCoefficients.mat', 'c_coefficients')
```

B Ex2

```
%% Setup
clc
close all
clear variables

resolution = 64; maxTime = 32;
signalLength = 64*32;

time = (1:signalLength)./resolution;
```

```
%% Find the Dual basis of the B-Spline
syms z;
%construct P
subPoly = 2 - z - z^-1;
sum_P = 1 + subPoly + (5/8)*subPoly^2 + (5/16)*subPoly^3;
% comes from H(z) = P(z)/G(z)
H = expand(((z+1)^4) * (sum_P));
%sum2poly doesn't work on negative indices, so shift H up by 3
dual_phi_T = sym2poly((z^3)*H);

%% Calculate the B spline scaling function
[beta0, psi_T, xval] = wavefun('haar', 1);
phi_T = conv(beta0, conv(beta0,conv(beta0,conv(beta0, beta0))));
phi_T = phi_T(4:end-2); % cut off zero parts of function

figure
plot(phi_T,'LineWidth',2)
axis tight
xlabel('Tap')
ylabel('B3 Scaling Function')

%{
subplot(1,2,2)
plot(dual_phi_T,'LineWidth',2)
axis tight
xlabel('Tap')
title('Dual Basis of B3 Scaling Function')

%% Set Up Coefficient Computation

m_degree = 0:3;
n_numCoefficients = 1024;

t = ones(4,signalLength); % t^0 and initialise rest of matrix
t(2,:) = (0:signalLength-1)./64; % t^1
t(3,:) = t(2,:).^2; % t^2
t(4,:) = t(2,:).^3; % t^3
t(5,:) = t(2,:).^4; % t^4

%% Compute coefficients

c_coefficients = zeros(length(m_degree),n_numCoefficients);
for mIndex = m_degree+1
    for nIndex = 1:n_numCoefficients
        shift = (nIndex-1) * 2;% resolution; % find the shift
        phiShifted = zeros(1,signalLength); % initialise phiShifted
        phiShifted((1 + shift):(length(phi_T) + shift)) = dual_phi_T; % compute ...
            phiShifted
        phiShifted = phiShifted(1:signalLength); % crop to signalLength

        c_coefficients(mIndex, nIndex) = ...
            dot(t(mIndex,:),phiShifted)./(resolution*4); % inner product to find c
    end
end

%% Reproduce the polynomials

t_reproduced = zeros(length(m_degree),signalLength);
figure

for mIndex = m_degree+1
    title1 = sprintf('Reconstructed monomial, order = %i',mIndex-1);

    subplot(2,2,mIndex)
    hold on

```

```
    % compute and plot reconstruction
    for nIndex = 1:n_numCoefficients
        shift = (nIndex-1) * 2;%resolution; % find the shift
        phiShifted = zeros(1,signalLength); % initialise phiShifted
        phiShifted((1 + shift):(length(phi_T) + shift)) = phi_T; % compute phiShifted
        phiShifted = phiShifted(1:signalLength); % crop to signalLength

        reproduction_contribution = c_coefficients(mIndex,nIndex) * phiShifted; % ...
            compute contribution at this m and n
        t_reproduced(mIndex,:) = t_reproduced(mIndex,:) + ...
            reproduction_contribution; % add to sum
        %plot(time,reproduction_contribution)
        plot(reproduction_contribution)
    end
    %h1 = plot(time,t_reproduced(mIndex,:),'b','LineWidth',2);
    %h2 = plot(time,t(mIndex,:),'r--','LineWidth',2);
    h1 = plot(t_reproduced(mIndex,:),'b','LineWidth',2);
    h2 = plot(t(mIndex,:),'r--','LineWidth',2);
    legend([h1 h2],{'Reconstruction', 'Actual'}, 'Location', 'northwest')
    xlim([0 signalLength])
    xlabel('time \s')
    axis tight
    ylabel(title1)
end
%}
```

C Ex3

```
%% Setup
clc
close all
clear variables

load project_files_&_data/tau.mat

tau = tau';

resolution = 64; maxTime = 32;
signalLength = 64*32;
time = (1:signalLength)./resolution;

%% Apply Annihilating Filter Method

[ h, tk_locations_est, ak_weights_est, y ] = annihilatingFilterMethod(tau);

% initialise vector and add diracs
x_diracsStream_est = zeros(1,signalLength);
x_diracsStream_est(uint32(tk_locations_est(1)*resolution+1)) = ak_weights_est(1);
x_diracsStream_est(uint32(tk_locations_est(2)*resolution+1)) = ak_weights_est(2);

figure
stem(time,x_diracsStream_est,'x')
axis tight
ylabel('amplitude')
xlabel('time /s')
%title('Reconstruction of Dirac Stream')

disp('Estimated Dirac values -')
disp('Locations:')
disp(tk_locations_est)
disp('Weights:')
disp(ak_weights_est)
```

D annihilatingFilterMethod.m

```
function [ h_annihilatingFilter, tk_locations, ak_weights, y  ] = ...
    annihilatingFilterMethod(tau_moments)
%annihilatingFilterMethod
% calcualte annihilating filter values for signal
% and return locations and weights

K = 2;
N = length(tau_moments)-1;

%% Find annihilating filter by solving equation

h_annihilatingFilter = zeros(K+1,1);
h_annihilatingFilter(1) = 1;

eqn_tauVect1 = tau_moments(K+1:N+1); % column vector of tau from K to N
eqn_tauMatrix = zeros(N-K+1, K); % initialise N-K x K-1 matrix
for index = 0:K-1
    eqn_tauMatrix(:,K-index) = tau_moments(1+index : N-K+1+index);
end

% solve eqn_tauMatrix * eqn_hVect = eqn_tauVect1;
%eqn_hVect = eqn_tauMatrix^-1 * -eqn_tauVect1;
eqn_hVect = eqn_tauMatrix \ -eqn_tauVect1;

h_annihilatingFilter(2:end) = eqn_hVect; % solve equation to find filter values

%% Find convolution

y = conv(tau_moments,h_annihilatingFilter);

%% Find locations

tk_locations = roots(h_annihilatingFilter); % locations are roots of filter

% Print location values
%disp('Locations:')
%disp(tk_locations.')

%% Solve Vandermonde system to find weights

eqn_locationsMatrix = [ 1 1; tk_locations(1) tk_locations(2) ];
eqn_tauVect2 = tau_moments(1:K);

ak_weights = eqn_locationsMatrix \ eqn_tauVect2; % solve equation to find weights

end
```

E Ex4

```
%% Setup
clc
close all
clear variables

load reproductionCoefficients.mat

K = 2;

```

```matlab
resolution = 64; maxTime = 32;
signalLength = 64*32;
time = (1:signalLength)./resolution;

%% Create stream of Diracs

ak_weights = [5; 2];
tk_locations = [12.5; 23];

% initialise vector and add diracs
x_diracsStream = zeros(1,signalLength);
x_diracsStream(tk_locations(1)*resolution+1) = ak_weights(1);
x_diracsStream(tk_locations(2)*resolution+1) = ak_weights(2);

figure
subplot(2,1,1)
stem(time,x_diracsStream,'x')
axis tight
ylabel('amplitude')
xlabel('time /s')
title('Original Dirac Stream')
hold on

disp('Original Dirac values -')
disp('Locations:')
disp(tk_locations)
disp('Weights:')
disp(ak_weights)

%% Compute and plot Daubechies scaling function

% N ≥ 2K-1 (p70)
% N = 2*(2)-1 = 3
% N+1 = 3+1 = 4 -> db4

phi = zeros(1,signalLength);
[phi_T,¬,¬] = wavefun('db4',6);
phi(1:length(phi_T))=phi_T;

%% Sample signal using Daubechies scaling function

m_degree = 0:3;
n_numSamples = 32;

y_sampled = zeros(1, n_numSamples);
for sampleIndex = 1:n_numSamples
    shift = (sampleIndex-1) * resolution; % find the shift
    phiShifted = zeros(1,signalLength); % initialise phiShifted
    phiShifted((1 + shift):(length(phi_T) + shift)) = phi_T; % compute phiShifted
    phiShifted = phiShifted(1:signalLength); % crop to signalLength

    y_sampled(sampleIndex) = dot(x_diracsStream,phiShifted); % compute samples
end

% plot samples
%figure
%stem(y_sampled)

%% Retrieve N+1 moments of signal

s_moments = zeros(length(m_degree),1);
for degreeIndex = m_degree+1
    s_moments(degreeIndex) = dot(c_coefficients(degreeIndex,:),y_sampled);
end

%% Apply annihilating filter method

```

```
[h, tk_locations_est, ak_weights_est,y] = annihilatingFilterMethod(s_moments);

disp('Estimated Dirac values -')
disp('Locations:')
disp(tk_locations_est)
disp('Weights:')
disp(ak_weights_est)

% initialise vector and add diracs
x_diracsStream_est = zeros(1,signalLength);
x_diracsStream_est(uint32(tk_locations_est(1)*resolution+1)) = ak_weights_est(1);
x_diracsStream_est(uint32(tk_locations_est(2)*resolution+1)) = ak_weights_est(2);

%% Display and Plot Results

subplot(2,1,2)
stem(time,x_diracsStream_est,'x')
axis tight
ylabel('amplitude')
xlabel('time /s')
title('Reconstruction of Dirac Stream')

disp('Original Dirac values -')
disp('Locations:')
disp(tk_locations_est)
disp('Weights:')
disp(ak_weights_est)
```

F Ex5

```
%% Setup
clc
close all
clear variables

load reproductionCoefficients.mat
load project_files_&_data/samples.mat

K = 2;

resolution = 64; maxTime = 32;
signalLength = 64*32;
time = (1:signalLength)./resolution;

%% Retrieve N+1 moments of signal

m_degree = 0:3;

s_moments = zeros(length(m_degree),1);
for degreeIndex = m_degree+1
    s_moments(degreeIndex) = dot(c_coefficients(degreeIndex,:),y_sampled);
end

%% Apply annihilating filter method

[h, tk_locations_est, ak_weights_est, y] = annihilatingFilterMethod(s_moments);

%% Display and Plot Results

% initialise vector and add diracs
x_diracsStream_est = zeros(1,signalLength);
x_diracsStream_est(uint32(tk_locations_est(1)*resolution+1)) = ak_weights_est(1);
x_diracsStream_est(uint32(tk_locations_est(2)*resolution+1)) = ak_weights_est(2);
```

```
stem(time,x_diracsStream_est,'x')
axis tight
ylabel('amplitude')
xlabel('time /s')
title('Reconstruction of Dirac Stream')

disp('Estimated Dirac values -')
disp('Locations:')
disp(tk_locations_est)
disp('Weights:')
disp(ak_weights_est)

%% Compute and plot Daubechies scaling function

% N ≥ 2K-1 (p70)
% N = 2*(2)-1 = 3
% N+1 = 3+1 = 4 -> db4

phi = zeros(1,signalLength);
[phi_T,¬,¬] = wavefun('db4',6);
phi(1:length(phi_T))=phi_T;

%% Sample signal using Daubechies scaling function

m_degree = 0:3;
n_numSamples = 32;

y_sampled2 = zeros(1, n_numSamples);
for sampleIndex = 1:n_numSamples
    shift = (sampleIndex-1) * resolution; % find the shift
    phiShifted = zeros(1,signalLength); % initialise phiShifted
    phiShifted((1 + shift):(length(phi_T) + shift)) = phi_T; % compute phiShifted
    phiShifted = phiShifted(1:signalLength); % crop to signalLength

    y_sampled2(sampleIndex) = dot(x_diracsStream_est,phiShifted); % compute samples
end

figure

subplot(2,1,1)
stem(y_sampled)
axis tight
ylabel('amplitude')
xlabel('time /s')
title('Original Samples')

subplot(2,1,2)
stem(y_sampled2)
axis tight
ylabel('amplitude')
xlabel('time /s')
title('Samples of Reconstruction')
```

G Ex6

```
%% Setup
clc
close all
clear variables

K = 2;

```

```
resolution = 64; maxTime = 32;
signalLength = 64*32;

%% Create stream of Diracs

ak_weights = [7; 4];
tk_locations = [7.5; 23];

% initialise vector and add diracs
x_diracsStream = zeros(1,signalLength);
x_diracsStream(tk_locations(1)*resolution+1) = ak_weights(1);
x_diracsStream(tk_locations(2)*resolution+1) = ak_weights(2);

figure
stem(x_diracsStream,'x')
xlim([0 signalLength])

disp('Original Dirac values -')
disp('Locations:')
disp(tk_locations')
disp('Weights:')
disp(ak_weights')

%% Daubechies
N_range = 5:8;
noiseSigmas = sqrt([0.001 0.01 0.1 1 10]);
% Multiplying by a gives variance of a^2.

for order = N_range
    % Create moments
    [moments, phi, ¬] = momentsGenerator(x_diracsStream, order);
    % Duplicate
    momentsNoise = repmat(moments.', length(noiseSigmas), 1);

    % Add different values of noise.
    for index = 1:length(noiseSigmas)
        momentsNoise(index,:) = noiseSigmas(index) * randn(1, length(moments)) + ...
            momentsNoise(index,:);
    end

    % Picture time!
    figure('position',[0 0 1280 800]);

    subplot(3, 2, 1);
    plot(phi);
    axis([0 7*resolution -0.4 1.2]);
    the_title = ['Order ' num2str(order) ' Daubechies Sampling Kernel'];
    title(the_title);
    xlabel('Time');

    for index = 1:length(noiseSigmas)
        subplot(3, 2, index+1);
        stem(momentsNoise(index, :),'x');
        xlabel('m');
        title(['Moment, \sigma^2: ' num2str(noiseSigmas(index)^2)]);
    end

    filename = ['noisyMoments' num2str(order)];
    save(filename, 'momentsNoise', 'phi');
end
```

H momentsGenerator.m

```
function [ moments, phi, coefficients ] = momentsGenerator( x_diracsStream, ...
    degree )
%DAUBECHIEMOMENTS Creates moments of function with Daubechie of degree.
%   Makes Q7 easier with different N

    % Generate the Daubechie scaling function
    signalLength = length(x_diracsStream);
    dBOrder = num2str(degree);
    wavetype = ['db' dBOrder];
    phi = zeros(1, signalLength);
    [phi_T, ¬, ¬] = wavefun(wavetype, 6);
    phi(1:length(phi_T))=phi_T;

    n_numVectors = 0:31;
    resolution = 64;

    % Create t
    % Construct the vectors of t^m, m = 0:degree-1 .
    tVals = ones(signalLength, degree);
    for order = 2:degree
        tVals(:, order) = ((0:signalLength-1)/resolution).^(order - 1);
    end

    % Create shifted phi
    allPhi = zeros(length(phi),length(n_numVectors));
    for n = n_numVectors
        allPhi(:,n+1) = [zeros(1, n*resolution) phi(1:end - n*resolution)];
    end

    % Acquire coefficients
    coefficients =  (tVals.' * allPhi)./resolution;

    % Take xt and make yn
    yn =  x_diracsStream * allPhi;
    % Moments
    moments = coefficients * yn.';

end
```

I Ex7 with No Enhancements

```
%% Setup
clear;
close all;
clc;

% Some constants
resolution = 64;
maxTime = 32;
signalLength = resolution * maxTime;
numberOfIterations = 6;
K = 2;
time = (1:signalLength)./resolution;

%% Create stream of Diracs

ak_weights = [7; 4];
tk_locations = [7.5; 23];

% initialise vector and add diracs
x_diracsStream = zeros(1,signalLength);
x_diracsStream(tk_locations(1)*resolution+1) = ak_weights(1);
```

```
x_diracsStream(tk_locations(2)*resolution+1) = ak_weights(2);

%% Acquire moments and Calculate Diracs
for moment = 5:8
    if moment == 5
        load('noisyMoments5.mat');
    elseif moment == 6
        load('noisyMoments6.mat');
    elseif moment == 7
        load('noisyMoments7.mat');
    elseif moment == 8
        load('noisyMoments8.mat');
    end
    noiseSigmas = size(momentsNoise, 1);

    % Calculate the Diracs
    for noiseIndex = 1:noiseSigmas
        tau_moments = momentsNoise(noiseIndex, :);
        [h, tk_locations_est, ak_weights_est] = ...
            annihilatingFilterMethod(tau_moments');

            x_diracsStream_est = zeros(1,signalLength);
            for diracIndex = 1:K
                x_diracsStream_est(uint32(tk_locations_est(diracIndex)*resolution+1)) ...
                    = ak_weights_est(diracIndex);
            end

            figure

            stem(time, x_diracsStream_est,'x');
            axis tight
            title('Reconstructed Signal');
            xlabel('time');
    end
end
```

J Ex7 with LTS

```
%% Setup
clear;
close all;
clc;

% Some constants
resolution = 64;
maxTime = 32;
signalLength = resolution * maxTime;
numberOfIterations = 6;
K = 2;
time = (1:signalLength)./resolution;

%% Create stream of Diracs

ak_weights = [7; 4];
tk_locations = [7.5; 23];

% initialise vector and add diracs
x_diracsStream = zeros(1,signalLength);
x_diracsStream(tk_locations(1)*resolution+1) = ak_weights(1);
x_diracsStream(tk_locations(2)*resolution+1) = ak_weights(2);

%% Acquire moments and Calculate Diracs
for moment = 5:8
```

```
    if moment == 5
        load('noisyMoments5.mat');
    elseif moment == 6
        load('noisyMoments6.mat');
    elseif moment == 7
        load('noisyMoments7.mat');
    elseif moment == 8
        load('noisyMoments8.mat');
    end
    noiseSigmas = size(momentsNoise, 1);

    % Calculate the Diracs
    for noiseIndex = 1:noiseSigmas
        tau_moments = momentsNoise(noiseIndex, :);
        [h, tk_locations_est, ak_weights_est] = ...
            annihilatingFilterMethodTLS(tau_moments);

            x_diracsStream_est = zeros(1,signalLength);
            for diracIndex = 1:K
                x_diracsStream_est(uint32(tk_locations_est(diracIndex)*resolution+1)) ...
                    = ak_weights_est(diracIndex);
            end

            figure

            stem(time, x_diracsStream_est,'x');
            axis tight
            title('Reconstructed Signal');
            xlabel('time');
    end
end
```

K annihilatingFilterMethodTLS.m

```
function [ h_annihilatingFilter, tk_locations, ak_weights ] = ...
    annihilatingFilterMethodTLS( tau_moments )

    K = 2;
    N = length(tau_moments) - 1;
    % Construct S
    c = tau_moments(K+1:N);
    r = fliplr(tau_moments(1:K+1));
    S = toeplitz(c', r);
    % SVD to get V
    [¬, ¬, V] = svd(S);
    % H is last column of V
    h_annihilatingFilter = V(:, end);
    % As before
    tk_locations = roots(h_annihilatingFilter);

    % Solve the vandermonde system is:
    eqn_locationsMatrix = ones(K,K);
    for rowIndex = 2:K
        eqn_locationsMatrix(rowIndex,:) = tk_locations.' .^(rowIndex-1);
    end
    eqn_tauVect2 = tau_moments(1:K)';

    ak_weights = eqn_locationsMatrix \ eqn_tauVect2; % solve equation to find weights

end
```

L Ex7 with LTS and Cadzow

```
%% Setup
clear;
close all;
clc;

% Some constants
resolution = 64;
maxTime = 32;
signalLength = resolution * maxTime;
numberOfIterations = 6;
K = 2;
time = (1:signalLength)./resolution;

%% Create stream of Diracs

ak_weights = [7; 4];
tk_locations = [7.5; 23];

% initialise vector and add diracs
x_diracsStream = zeros(1,signalLength);
x_diracsStream(tk_locations(1)*resolution+1) = ak_weights(1);
x_diracsStream(tk_locations(2)*resolution+1) = ak_weights(2);

%% Acquire moments and Calculate Diracs
for moment = 5:8
    if moment == 5
        load('noisyMoments5.mat');
    elseif moment == 6
        load('noisyMoments6.mat');
    elseif moment == 7
        load('noisyMoments7.mat');
    elseif moment == 8
        load('noisyMoments8.mat');
    end
    noiseSigmas = size(momentsNoise, 1);

    % Calculate the Diracs
    for noiseIndex = 1:noiseSigmas
        tau_moments = momentsNoise(noiseIndex, :);
        [h, tk_locations_est, ak_weights_est] = ...
            annihilatingFilterMethodCadzow(tau_moments, 10);

            x_diracsStream_est = zeros(1,signalLength);
            for diracIndex = 1:2
                x_diracsStream_est(uint32(tk_locations_est(diracIndex)*resolution+1)) ...
                    = ak_weights_est(diracIndex);
            end

            figure

            stem(time, x_diracsStream_est,'x');
            axis tight
            title('Reconstructed Signal');
            xlabel('time');
    end
end
```

M annihilatingFilterMethodCadzow.m

```matlab
function [ h_annihilatingFilter, tk_locations, ak_weights ] = ...
    annihilatingFilterMethodCadzow( tau_moments, iterations )

    K = 2;
    N = length(tau_moments) - 1;
    % Construct S
    c = tau_moments(K+1:N);
    r = fliplr(tau_moments(1:K+1));
    S = toeplitz(c', r);
    % SVD to get V
    for iteration = 1:iterations
        [U, D, V] = svd(S);
        % Remove smaller eigenvalues
        Dr = size(D, 1) - K;
        Dc = size(D, 2) - K;
        Ddash = [D(1:K, 1:K), zeros(K, Dc); zeros(Dr, K), zeros(Dr, Dc)];
        S = U * Ddash * V';
        % Make "Toeplitz"
        Pr = size(S, 1);
        Pc = size(S, 2);

        if Pr > Pc
            maxSize = Pr;
        else
            maxSize = Pc;
        end

        S_temp = zeros(maxSize, maxSize);
        for dIndex = -(Pr-1):1:Pc-1
            working = diag(S, dIndex);
            length = maxSize - abs(dIndex);

            temp = mean(working);
            new = diag(temp * ones(length, 1), dIndex);
            S_temp = S_temp + new;
        end
        S = S_temp(1:Pr, 1:Pc);
    end
    [¬, ¬, V] = svd(S);
    % H is last column of V
    h_annihilatingFilter = V(:, end);
    % As before
    tk_locations = roots(h_annihilatingFilter);

    % Solve the vandermonde system is:
    eqn_locationsMatrix = ones(K,K);
    for rowIndex = 2:K
        eqn_locationsMatrix(rowIndex,:) = tk_locations.' .^(rowIndex-1);
    end
    eqn_tauVect2 = tau_moments(1:K)';

    ak_weights = eqn_locationsMatrix \ eqn_tauVect2; % solve equation to find weights

end
```