LAWRENCE LIVERMORE NATIONAL LABORATORY

LLNL-TR-524752

Noise Reduction with Microphone Arrays for Speaker Identification

Z. Cohen

January 27, 2012

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Noise Reduction with Microphone Arrays for Speaker Identification

Zack Cohen
Graduate Intern
Lawrence Livermore National Laboratory

June 2011 – December 2011

Introduction

Reducing acoustic noise in audio recordings is an ongoing problem that plagues many applications. This noise is hard to reduce because of interfering sources and non-stationary behavior of the overall background noise. Many single channel noise reduction algorithms exist but are limited in that the more the noise is reduced; the more the signal of interest is distorted due to the fact that the signal and noise overlap in frequency.

Specifically acoustic background noise causes problems in the area of speaker identification. Recording a speaker in the presence of acoustic noise ultimately limits the performance and confidence of speaker identification algorithms. In situations where it is impossible to control the environment where the speech sample is taken, noise reduction filtering algorithms need to be developed to clean the recorded speech of background noise. Because single channel noise reduction algorithms would distort the speech signal, the overall challenge of this project was to see if spatial information provided by microphone arrays could be exploited to aid in speaker identification.

Goals

The problems stated above motivated a project with the following goals:

- Test the feasibility of using microphone arrays to reduce background noise in speech recordings
- Characterize and compare different multichannel noise reduction algorithms
- Provide recommendations for using these multichannel algorithms
- Ultimately answer the question: Can the use of microphone arrays aid in speaker identification?

Delay Sum Beamforming

The first multichannel noise reduction approach that was looked at was the simple Delay and Sum beamformer. As its name suggests, this technique reduces noise by first delaying the signals at each microphone in the array according to their relative position and direction of arrival of the signal of interest. The next step is to add these aligned signals together to form a single “beamformed” output. These two steps effectively add the signal of interest coherently while adding interfering sources and background noise incoherently resulting in noise reduction.

When analyzing the Delay Sum beamformer it is often useful to look at its directional response. A directional response shows how signals from all directions contribute to the overall output of the spatial filter. The directional responses of different delay sum beamformers were calculated and plotted using “dir_response.m” and “DS_lin_dir_resp.m” for arbitrary arrays and linear arrays respectively. It was seen from these plots that once the time delays are set, the look direction of the beamformer is fixed and all signals from the look direction are passed while all others are attenuated according to the side lobe characteristics (Figure 1). Different array geometries specified in “Array_Geometry.m” were also tested. It was seen that the directional response of the beamformer was greatly affected by different geometries.

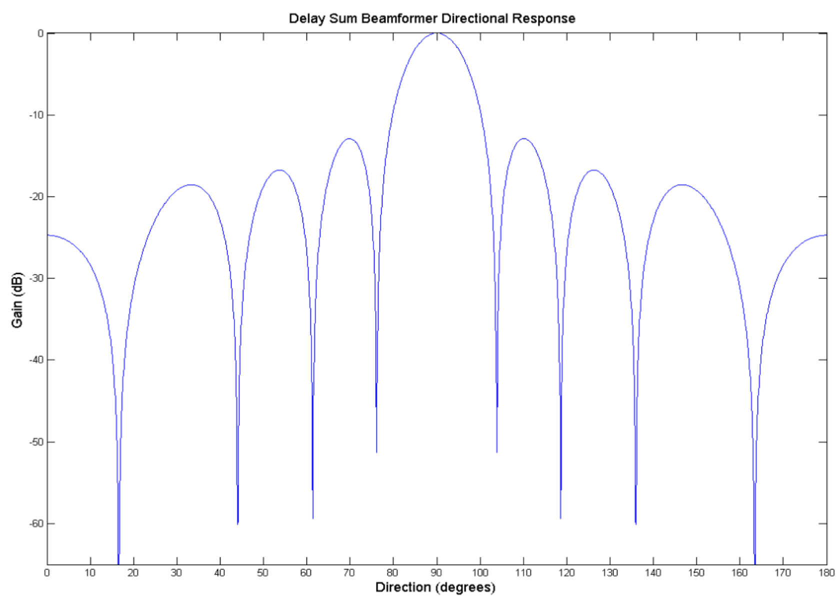


Figure 1

After looking at the theoretical directional response of the delay sum beamformer, it was of interest to create a simulator to see how the beamformer actually behaved. The m file “sim_3D.m” simulates a source signal at a certain position in space by delaying the signal corresponding to the position of each receiver in the array. The function “DS_beamformer.m” actually implements the delay and sum beamformer by correcting the delays caused by the simulator. By simulating a sinusoidal source from different directions and applying the delay sum beamformer with a fixed look direction, we were able to verify the theoretical directional response (Figure 2). The script “Init_3D_sim.m” was used to set up parameters for simulations such as source position, source signal and receiver positions.

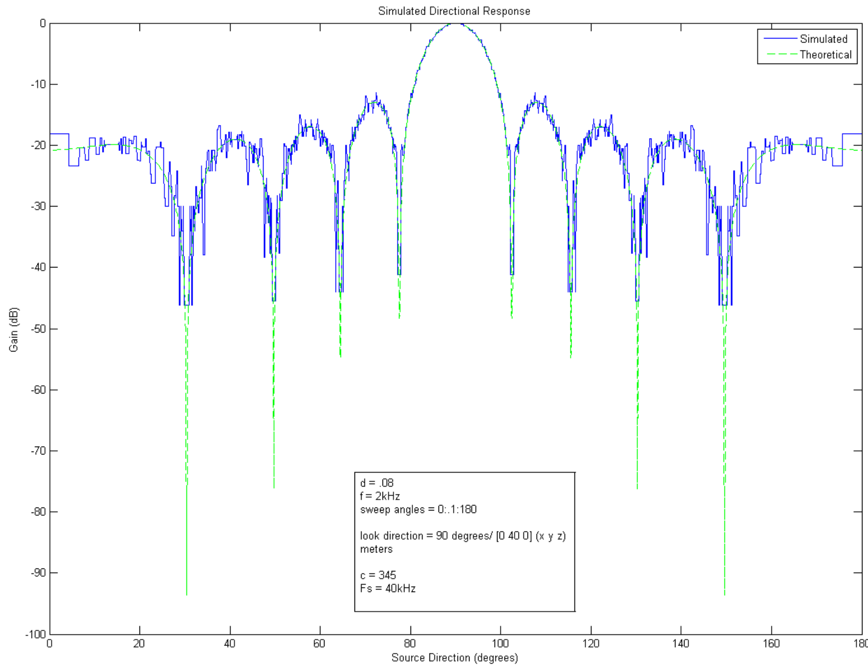


Figure 2

The delay sum beamformer seems like an adequate and simple solution to reducing noise by just delaying and summing signals. Problems arise with this approach when using it for broadband signals such as speech (300Hz-3kHz). These come from the fact that the directional response of the delay sum beamformer is frequency dependant. To further explore this, the directional response was plotted over a range of frequencies using “DS_freq_resp.m”. The plots generated by this function confirm the delay sum beamformer’s frequency dependence.

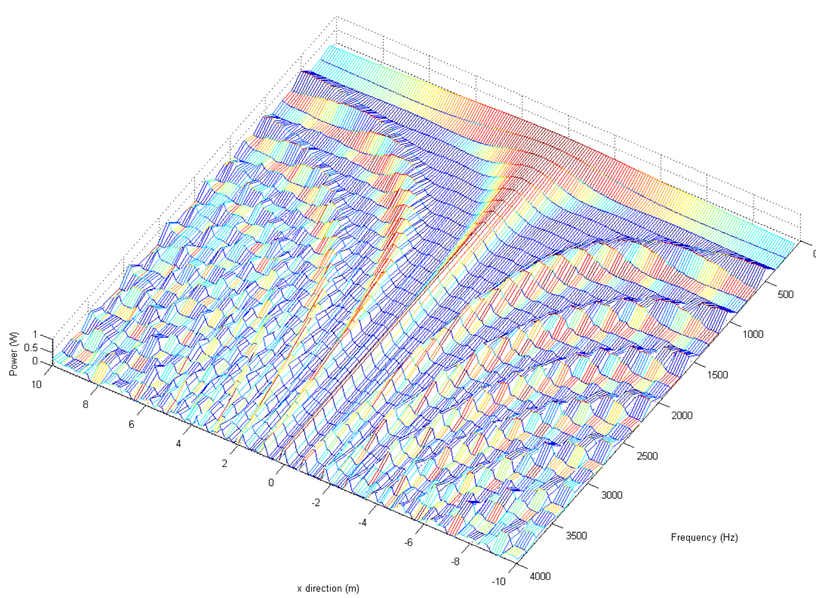


Figure 3

Figure 3 illustrates the problems that can arise from using the delay sum beamformer with broadband signals such as speech. Some of these problems include spatial aliasing and lowpass filtering from focusing error.

Another issue with the delay sum beamformer is that it is a fixed beamformer, meaning that it performs differently depending on how the background noise is correlated. To investigate this, a 2 dimensional noise model was derived to see how the delay sum beamformer was affected by spatially correlated noise. The following correlation was found:

$$\mathrm{Corr}(kr) = J_0(kr)\cos(2\pi f(t_m - t_n))$$

$$\mathbf{k} = \text{wave number} = 2\pi f/c$$

$$\mathbf{r} = \text{distance between mics 'm' and 'n'}$$

$$\mathbf{t_m - t_n} = \text{difference in applied time delays on channel 'm' and 'n'}$$

Using the function “plane_noise_R.m” the noise with the above correlation was simulated. The function “sim_steered_resp.m” sweeps the beamformer look direction and calculates the output power of the beamformer. Using the script “DS_noise_tests.m” to set up parameters, different frequency noise was generated and run through the delay sum beamformer with different look directions (Figure 4).

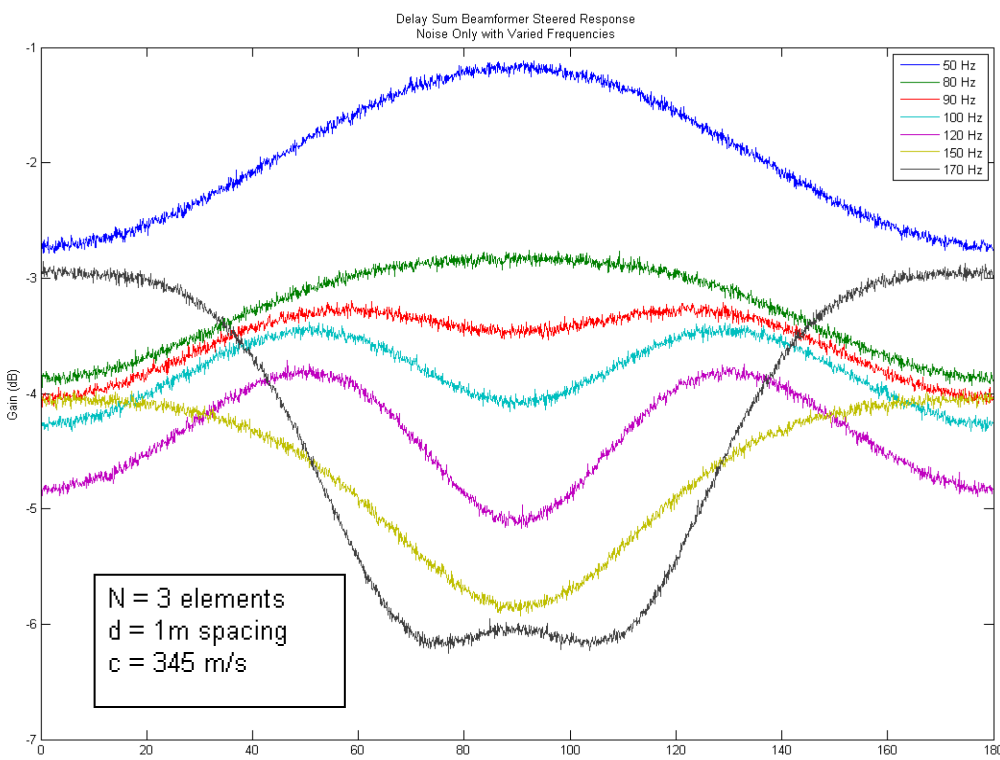


Figure 4

Figure 4 shows that lower frequency noise is highly correlated and creating peaks in the noise power when the beamformer is steered to 90 degrees. This phenomenon illustrates how the delay sum beamformer may not achieve as much noise reduction as expected for different look directions. Spatially correlated noise may also spoof the delay sum beamformer into thinking there is a source at 90 degrees if you are trying to track sources.

Other Narrowband Beamformers

The delay sum beamformer is considered a narrowband beamformer as its directional response is frequency dependant. Therefore it is most useful to design the delay sum beamformer for a single operating frequency so its directional response is known.

From the delay sum beamformer, other narrowband beamformers have been developed using the same delaying technique to control the look direction while paying more attention to channel weighting. By weighting the channels differently, the mainlobe and side lobe characteristics of the directional response are able to be controlled. The algorithms for calculating the channel weighting can be fixed or adaptive, making for many possibilities depending on applications.

To investigate these adaptive weighting techniques, the Maximum SNR beamformer was explored. This beamformer allows nulling of competing sources to achieve maximum signal to noise ratio. Given the estimated correlation structure of the noise, the Maximum SNR beamformer weights the different channels of the array to place a null in the direction of competing sources.

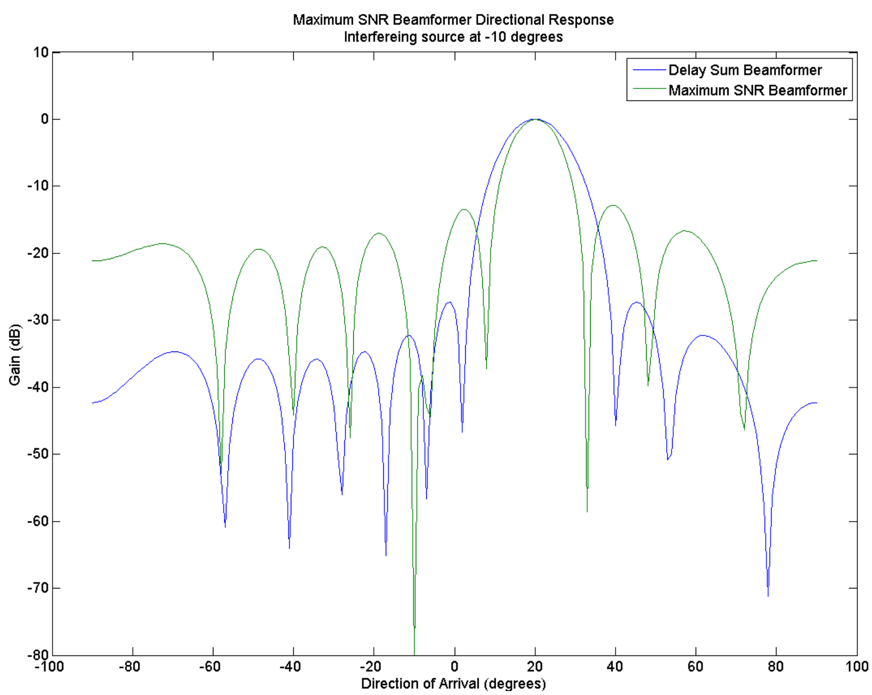


Figure 5

Figure 5 shows an example of a Maximum SNR directional response where the direction of arrival of the interfering source is at -10 degrees. The corresponding delay sum beamformer directional response is also plotted. Other narrowband techniques explored include Least Squared, Minimum Variance Distortionless Response (MVDR), Wiener Beamformers.

Exploring Narrowband beamforming is useful to gaining insight into array processing and adaptive filtering. These techniques are great for single frequency operation and are used in many applications such as radar, sonar and communications.

Broadband Beamforming

As discussed earlier the delay sum beamformer is a simple way to achieve noise reduction but it suffers when applied to the broadband case due to its frequency dependence. To process speech signals it is useful to develop and explore multichannel broadband filtering techniques.

The two chosen techniques for this project were the Multichannel Wiener Filter and the Spatio-Temporal Prediction Filter, both adaptive filters. The overall goal of both of these techniques is to recover the speech signal recorded at microphone 1 using all of the observation signals (speech + noise) recorded at all other microphones. This approach to noise reduction is extremely powerful in that the position of the microphones and speaker of interest need not be known. This information is implicitly calculated in the algorithms through space-time correlation matrices.

The Wiener filter is a classical optimal adaptive filter and is basically the well known single channel Wiener Filter expanded to the multichannel case. The algorithm used to calculate the filter coefficients works by minimizing the mean squared error of the estimated speech signal and the actual speech signal. Some of the correlation matrices calculated in this algorithm need information about the speech signal which is not known but can be estimated using speech + noise intervals and noise only intervals.

The Spatio-Temporal Prediction filter is also an optimal adaptive fitler and is similar to the well known LCMV filter. This filter exploits the idea that speech is partially predictable in time and space because it comes from a unique source. To calculate the filter coefficients, the algorithm first assumes there is some prediction matrix that estimates the speech signal at any microphone from the speech signal at microphone 1. Assuming this is true, the STP approach looks to minimize the output noise power with the constraint that the speech signal is not distorted. The optimal prediction matrix mentioned above is calculated by minimizing the error caused by using this prediction matrix to estimate all other speech signals.

Both of these broadband multichannel filters were implemented in matlab for utilization in data post processing. The filtering functions “Weiner_filter.m” and “Spatio_Temporal_Filter.m” apply the above filtering operations given ‘N’ channels of recorded noisy speech and ‘N’ channels of noise recorded when no speech is present. Both functions also allow you to specify filter length and overlap for tuning.

Simulations

To evaluate how these techniques performed with different known input parameters, a 3 dimensional spatially correlated noise model was derived:

$$R(r) = \int_0^{\omega c} \frac{\sin(kr)}{kr} \, d\omega$$

$$\boxed{R(r) = \frac{Si(k_c r)}{k_c r}}$$

where:

$$Si(z) = \int_0^{z} \frac{\sin(t)}{t} \, dt$$

$$k_c = \frac{\omega_c}{c}$$

$$\omega_c = 2\pi f_c = noise\ bandwidth$$

$$c = 345 \frac{m}{s} = speed\ of\ sound$$

Cook, Richard, R.V. Waterhouse , R.D Berendt, Seymour Edelman, and M.C. Thompson, Jr. "Measurement of Correlation Coefficients in Reverberant Sound Fields." *Journal of the Acoustical Society of America*. 27.6 (1955): 1072-77

With this noise model, simulated noise was generated with different bandwidths using "plane_noise_R_3D.m" and passed through both filters with varied filter lengths. Metrics such as Output SNR and Noise Reduction factor were calculated using the matlab function "BB_Filter_Metrics.m" for each filter. The matlab scripts "Wiener_Simulation_test1.m" was used to set up the simulation for both broadband filters while "DS_Simulation1_noiseBW.m" was used to simulate the delay sum beamformer.

Figure 6 and 7 below show the output SNR results for both broadband filters when the bandwidth of the noise and the filter length were varied. The Wiener filter performs fairly consistently over filter length, increasing and leveling off, but achieves moderate output SNR. The output SNR decreases from 300 Hz to 800 Hz noise bandwidth but increases above 11 dB for a 3kHz noise bandwidth. The STP filter achieves much higher output SNR for lower noise bandwidths but is sporadic over filter length. As the noise bandwidth increases, the output SNR decreases to about 10dB at 3kHz but has more predictable performance over filter length.

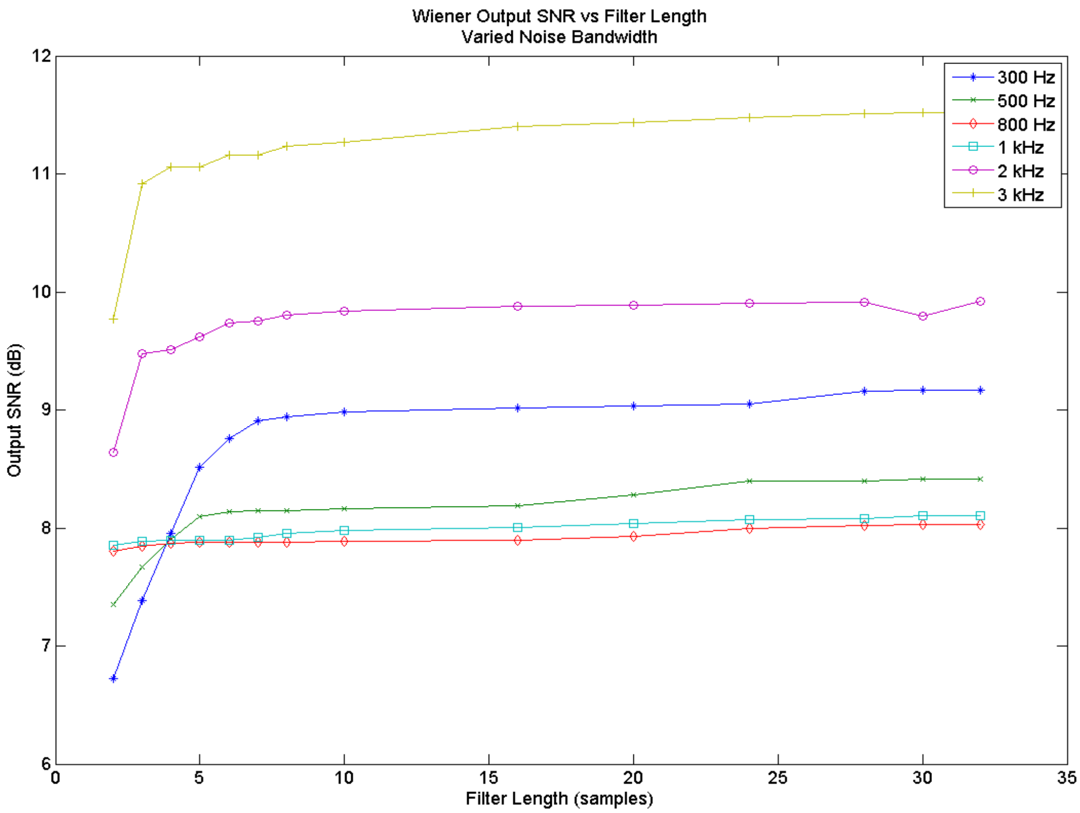


Figure 6

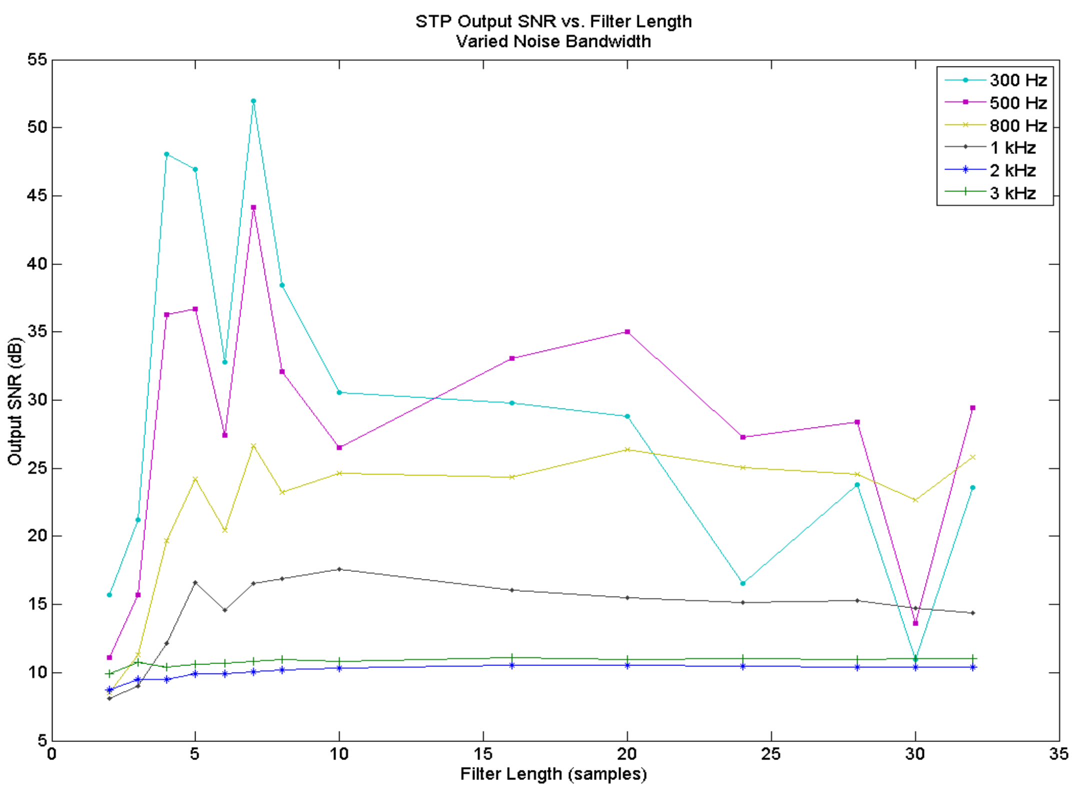


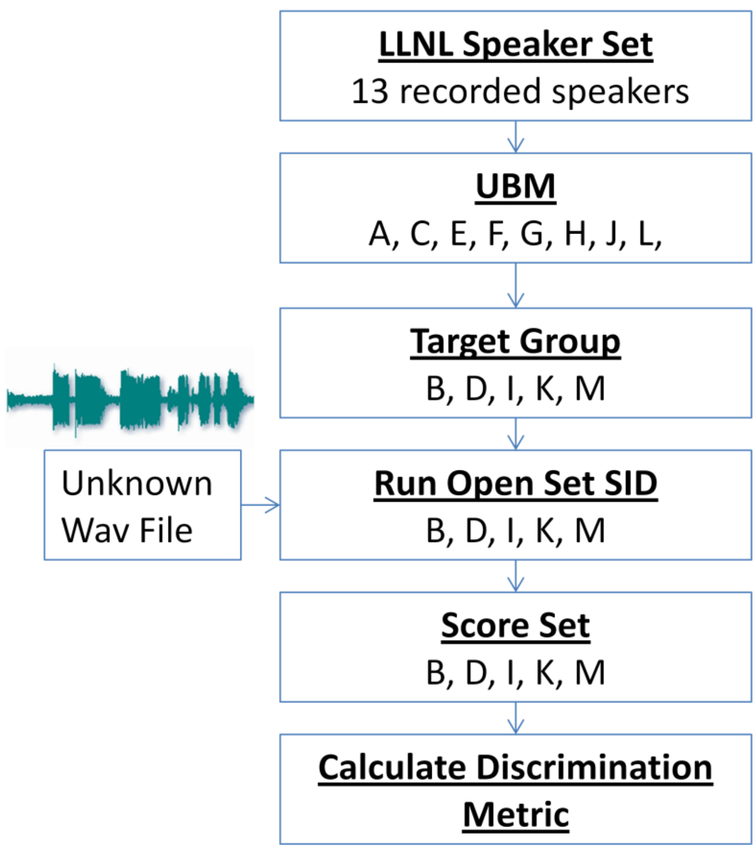
Figure 7

Speaker Identification

As the overall goal of the project is to see if microphone arrays can aid in speaker identification, it is useful to come up with a Speaker Identification Metric to judge these approaches. For this task we used RAPT-R, an audio processing tool developed by the Air Force Research Laboratory. Specifically, we utilized their speaker identification plug-in and their “Open-Set SID 2010” Speaker Identification algorithm which they recommended.

The process for calculating this SID metric is as follows:

Given an unknown speaker audio file and a Target group of “suspects”, run the unknown speaker file against the target group using the Open-Set SID 2010 algorithm. The algorithm gives a quantitative score of how likely each speaker in the target group is to be the unknown speaker. From this score, a more useful metric was calculated by taking the score of the actual speaker in the unknown file and subtracting the score of the next highest speaker from it.



SID Discrimination Metric

Score = Actual Speaker Score – Highest Score of other Speakers

Experiments and Data Collections

To evaluate the performance of these filters in real scenarios, a data collect was done outdoors using LLNL's microphone array. The data collect took place in the northwest perimeter of the lab near the corner of Vasco and Patterson Road on May 10th 2011. Two speaker audio files, male and female, were played back at a moderate volume level and their position was varied straight in front of the array (Figure 8 and 9). The ultimate goal of this collect was to see how far a positive SID could be achieved.

Figure 8

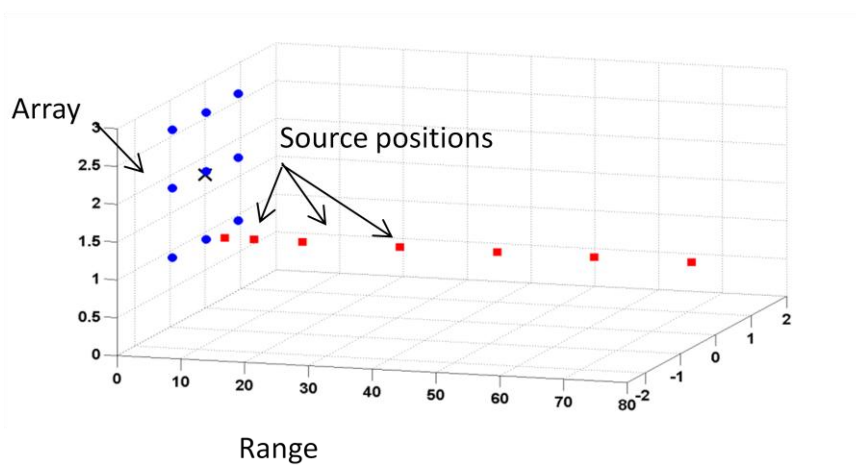


Figure 9

Results

The data from the outdoor collect was post-processed using the Delay Sum Beamformer, Multichannel Wiener Filter and Spatio-Temporal Prediction Filter. The filter lengths of the two broadband filters were varied to see how it affected the output SNR. The matlab scripts “Weiner_Test1_Filt_length.m”, “Weiner_Test2_overlap.m”, “Weiner_Test3_filt_length_max_overlap.m”, “Weiner_Test4_N.m”, “ST_Test1_Filt_length.m”, “ST_Test2_overlap.m”, “ST_Test3_filt_length_max_overlap.m”and “ST_Test4_N.m” were used to vary filter parameters and measure performance for each case. The matlab function “DS_metrics.m” and the script “DS_metrics_sweep.m” ran the delay sum beamformer on all cases of the data and calculated output metrics. The script “Preprocess_data.m” preprocessed the data from the collect before the filters were run.

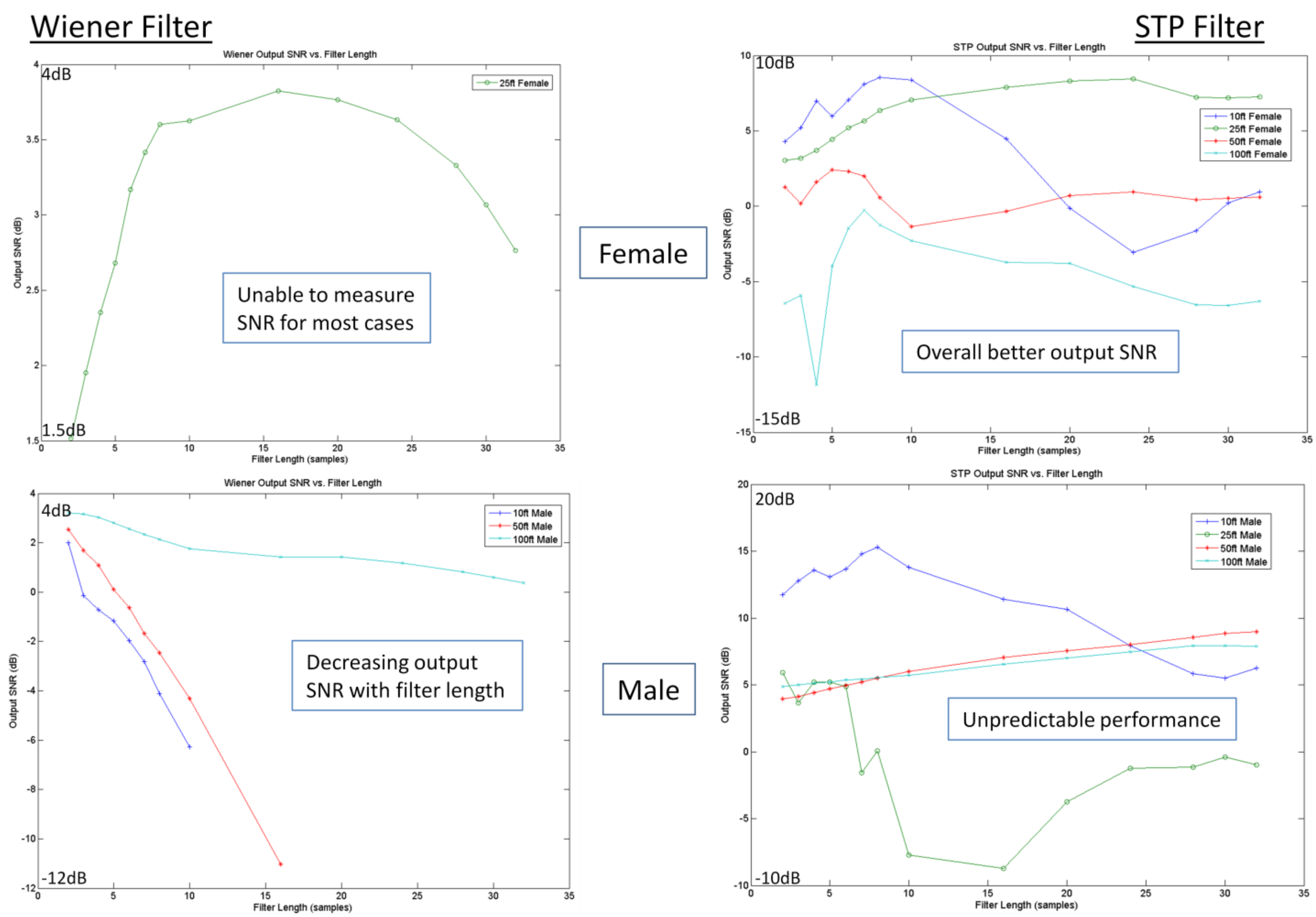
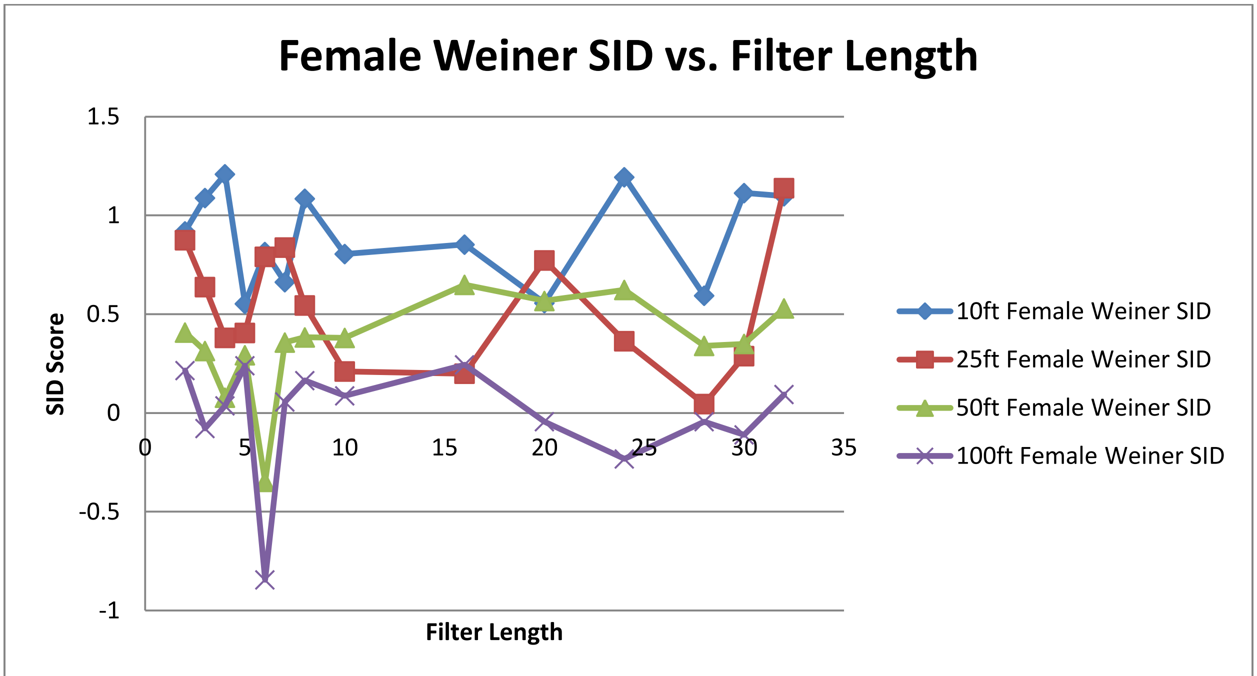
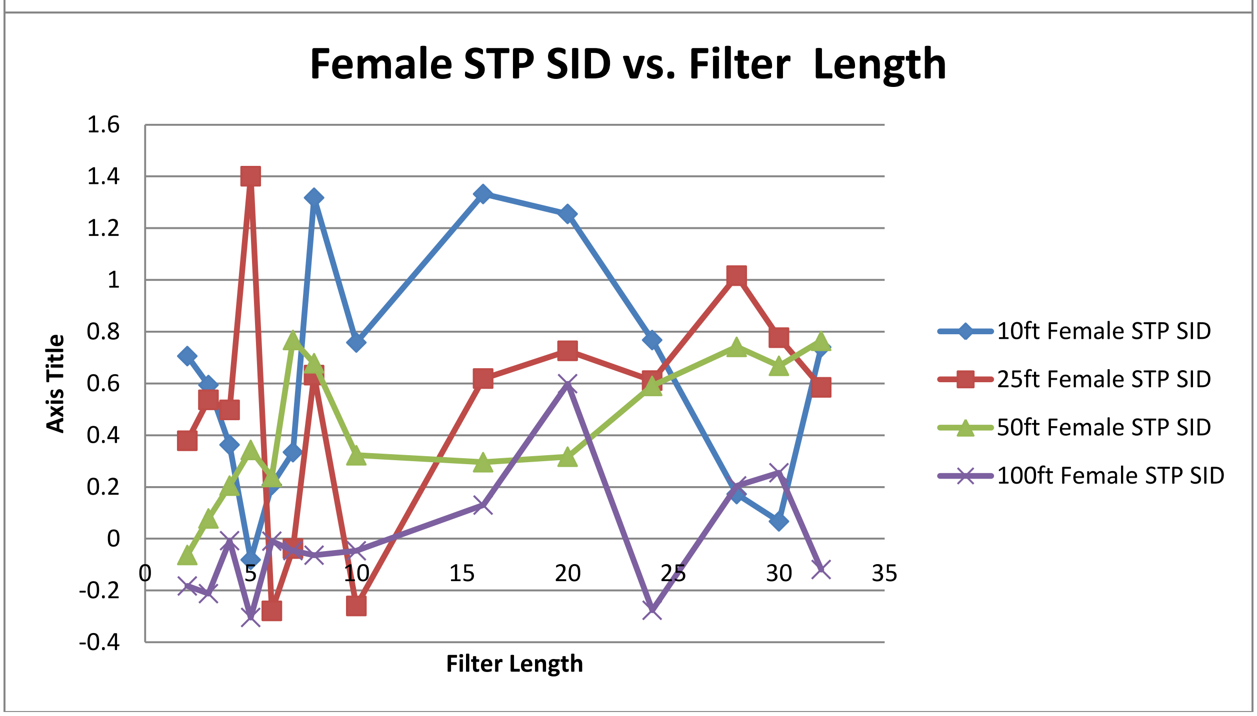


Figure 10

Figure 10 shows the output SNR achieved by both Filters for both male and female speakers for distances up to 100 ft. The Wiener Filter output SNR was difficult to measure and even decreased as filter length increased unlike the results of the simulations. The STP filter on the other hand had a lot better output SNR performance and was able to be measured for all cases. The output SNR trends seemed to vary greatly with speaker and position. Low input SNR and changing noise statistics over time seemed to be the reason for poor and unpredictable output SNR performance for both filters.

Despite inexplicable output SNR performance both filters did fairly well in terms of SID scoring.





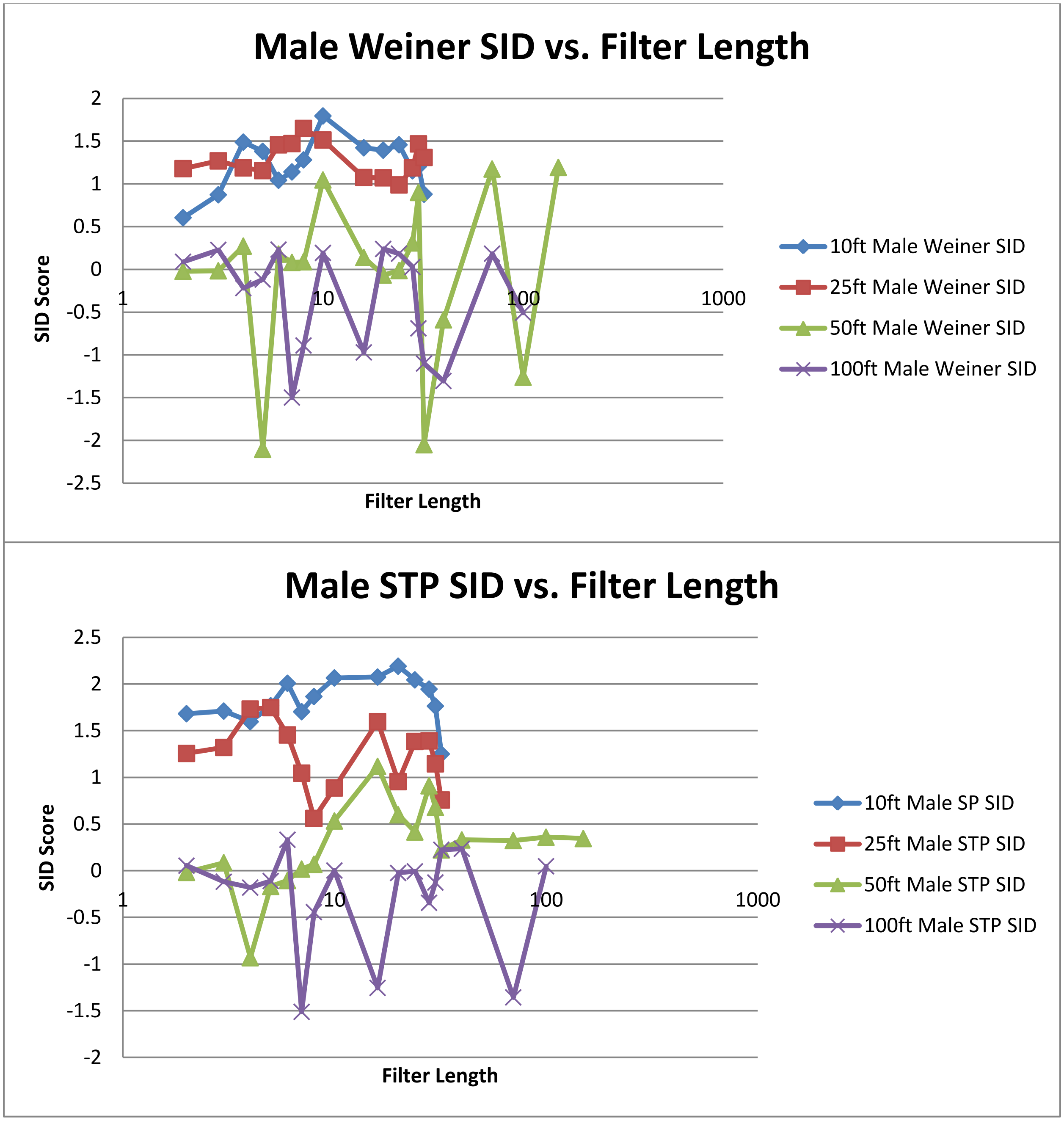
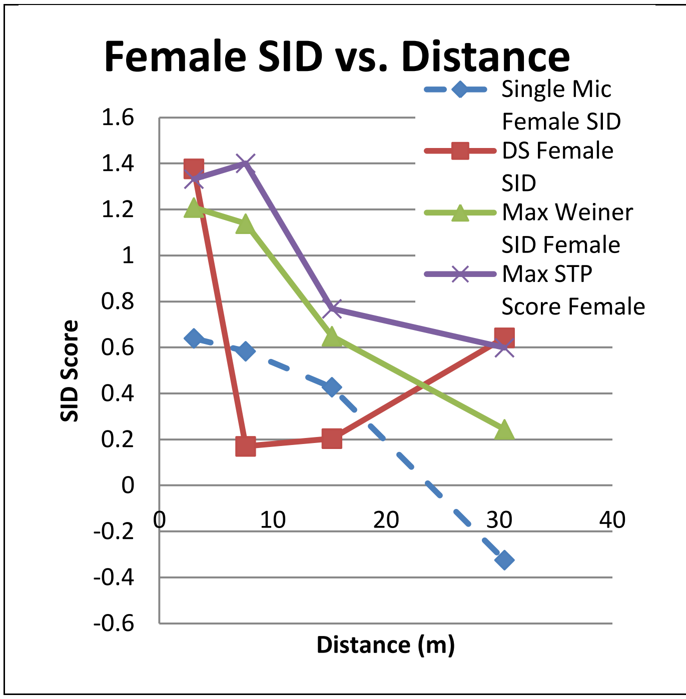
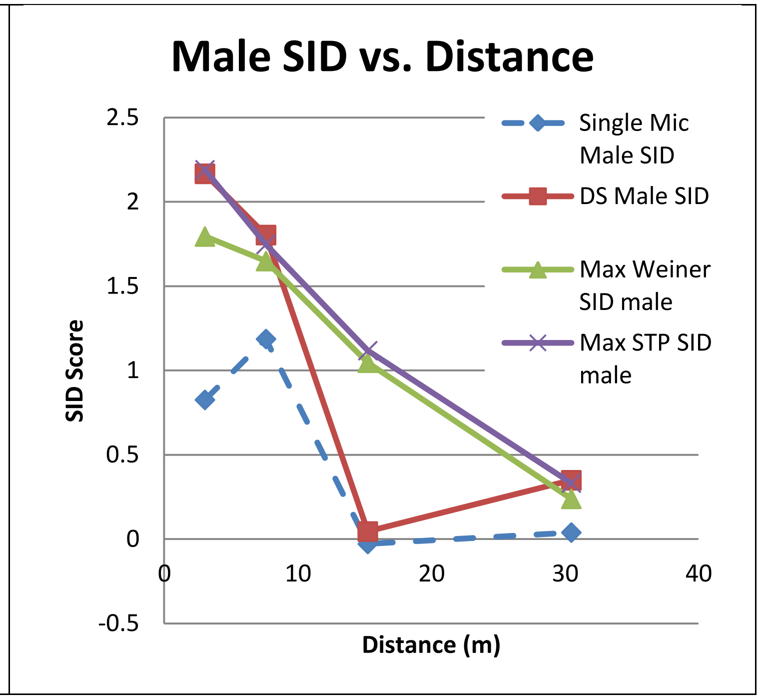


Figure 11

As seen in Figure 11 it's hard to extract trends in terms of filter length, output SNR and SID score. The SID scores seem to vary randomly with speaker as well as distance. When looking at the maximum scores achieved for each distance for each filter (Figure 12), it is easy to see that the STP filter performs the best, then the Wiener filter; with the Delay Sum beamformer scoring somewhat inconsistently for each distance. Overall, all 3 filters are able to hit positive SID scores at 100 ft where the single microphone cannot.





Conclusions

Overall the results from the outdoor data collect were not as clean as expected from the simulations. It seems as though it is important to have many trials and speakers to evaluate these techniques effectively. Speaker identification scoring seems to be dependant not only on SNR but speech distortion caused by the filters. This is an important conclusion because one would intuitively think that maximizing signal power would give the best SID but this is not always the case. Higher SNR does seem to help give the SID algorithm a better chance of achieving a high score but should not be considered a performance predictor. All three of these techniques enabled 100ft positive SID scores and confirm that that microphone arrays can aid in identifying speakers even at long distances.

Out of all of the approaches used in this project, the STP filter performs the best in terms of SNR and SID. The Wiener filter seems sensitive in practice to tuning in regards to statistical estimation, filter length and number of microphones used. The Delay Sum beamformer is a simple solution and performs decently but it requires knowledge of relative microphone and speaker positions. Different filters can be chosen for different applications depending on resources available and desired specifications.

Appendix – Matlab Codes

Array_Geometry.m

```
%% .08m Spaced 10 Sensor 2D Linear Array (book)
prec = [ -.36 0 0 ;
-.28 0 0 ;
-.20 0 0 ;
-.12 0 0 ;
-.04 0 0 ;
.04 0 0 ;
.12 0 0 ;
.20 0 0 ;
.28 0 0 ;
.36 0 0 ];

%% 1m Spaced 2D linear array

prec = [ -4.5 0 0 ;
-3.5 0 0 ;
-2.5 0 0 ;
-1.5 0 0 ;
-.5 0 0 ;
.5 0 0 ;
1.5 0 0 ;
2.5 0 0 ;
3.5 0 0 ;
4.5 0 0 ];

%% Experimental 3D Array setup
prec = [ .956   0   .445;
         0      0   .458;
         -.915  0   .445;
         .956   0   1.368;
         0      0   1.358;
         -.915  0   1.278;
         .956   0   2.139;
         0      0   2.133;
         -.915  0   2.075];
%% .01m spaced linear array
prec = [ -.045 0 0 ;
-.035 0 0 ;
-.025 0 0 ;
-.015 0 0 ;
-.005 0 0 ;
.005 0 0 ;
.015 0 0 ;
.025 0 0 ;
.035 0 0 ;
.045 0 0 ];

%% .05m Spaced linear array
prec = [ -.225 0 0 ;
-.175 0 0 ;
```

```
-.125 0 0 ;
-.075 0 0 ;
-.025 0 0 ;
.025 0 0 ;
.075 0 0 ;
.125 0 0 ;
.175 0 0 ;
.225 0 0 ];

%% .02m spaced linear array
prec = [ -.09 0 0 ;
-.07 0 0 ;
-.05 0 0 ;
-.03 0 0 ;
-.01 0 0 ;
.01 0 0 ;
.03 0 0 ;
.05 0 0 ;
.07 0 0 ;
.09 0 0 ];

%% .015m spaced lin array
prec = [ -.0675 0 0 ;
-.0525 0 0 ;
-.0375 0 0 ;
-.0225 0 0 ;
-.0075 0 0 ;
.0075 0 0 ;
.0225 0 0 ;
.0375 0 0 ;
.0525 0 0 ;
.0675 0 0 ];

%% .012m Spaced Linear Array
prec = [ -.054 0 0 ;
-.042 0 0 ;
-.030 0 0 ;
-.018 0 0 ;
-.006 0 0 ;
.006 0 0 ;
.018 0 0 ;
.030 0 0 ;
.042 0 0 ;
.054 0 0 ];

%% 1m Spacing 3 Sensor linear array

prec = [-1 0 0 ;
        0 0 0;
        1 0 0];
%% .8m Spaced 3 Sensor linear array
prec = [-.8 0 0 ;
        0 0 0;
        .8 0 0];
%% .5 Spaced 3 Sensor linear array
prec = [-.5 0 0 ;
```

```
        0 0 0;
        .5 0 0];
 %% .3 Spaced 3 Sensor linear array
prec = [-.3 0 0 ;
        0 0 0;
        .3 0 0];
%% .2 Spaced 3 Sensor linear array
prec = [-.2 0 0 ;
        0 0 0;
        .2 0 0];
%% .7 Spaced 3 Sensor linear array
prec = [-.7 0 0 ;
        0 0 0;
        .7 0 0];
%% .6 Spaced 3 Sensor linear array
prec = [-.6 0 0 ;
        0 0 0;
        .6 0 0];
```

Array_preprocess.m

```
function [] = Array_preprocess(datadir,datafile,FSout,fcutoff)
% function [] = Array_preprocess(datadir,datafile,Fsout,fcutoff)
% This function preprocesses the array data, by downsampling and high pass
% filtering. Inputs are:
%      datadir: data directory
%      datafile: data file (including .mat extension)
%      FSout: final sampling frequency
%      fcutoff: cutoff frequency for detrending (high pass filter)
% Output file name is input file name with _PP appended.

    dfheader = datafile(1:(end-4));
    outfile = [dfheader '_PP.mat'];

    load(fullfile(datadir,datafile));
    FSin = 1/hDelta;
    ndet = 2*round(.5*FSin/fcutoff);

    [TimeDatadt,~] = detrend_filt(TimeData,ndet);
    TimeDataPP = fractional_downsample(TimeDatadt,FSin,FSout);
    [nt,~] = size(TimeDataPP);
    time = (0:(nt-1))'/FSout;
    FS = FSout;

    save(fullfile(datadir,outfile),'time','FS','TimeDataPP');

end
```

BB_Filter_Metrics.m

```
function [ z_v_filt, SNR_in, SNR_out, SSNR_out, nr_factor, sd_factor, P_out,
P_out_n, P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src, noise, L,overlap, H,
z_k )
%Calculates broadband filter performance metrics given noisy speech and
```

```
%noise only mic array outputs.
%   src = N rows/channels of noise+speech signal with any length
%   noise = N rows/channels of noise only signal with any length
%   L = length of frame
%   overlap = # of sample overlap of frame
%   H = calculated filter matrix
%   z_k = single channel beamformed filter output (1x..)

N = size(src,1); % Calculate number of mics in array



for i = 1:L-overlap:length(noise)

    if(L+i-1 <= length(noise)) %if current block will exceed length of input
array, break loop

    v_L = noise(:,i:L+i-1)'; % Take a block of L samples at starting at
current index i

    %Organize NxL matrix containing sample blocks into NLx1 matrix
    v_k = v_L(:);

    z_v_filt(i:i+L-1,1) = H*v_k;
    end
end

P_out_n_SEG = 0;
P_out_SEG = 0;
m = 0;
n = 0;
for i = 1:L-overlap:max(length(noise),length(src))

    if(L+i-1 <= length(z_v_filt)) %if current block will exceed length of
input array, break for loop
        m = m+1;
        P_out_n_SEG = ((m-1)/m)*P_out_n_SEG + var(z_v_filt(i:L+i-1))./m;
    end

    if(L+i-1 <= length(z_k))
        n = n+1;
        P_out_SEG = ((n-1)/n)*P_out_SEG + var(z_k(i:L+i-1))./n;
    end

end
SSNR_out = (P_out_SEG - P_out_n_SEG)./P_out_n_SEG;

P_out = var(z_k);
P_out_n = var(z_v_filt);
P_sig1 = var(src(1,:)) - var(noise(1,:));

SNR_in = P_sig1./var(noise(1,:));
```

```
SNR_out = (P_out./P_out_n)-1;

nr_factor = var(noise(1,:))./P_out_n;



P_out_sig1 = P_out - P_out_n;

sd_factor = abs(P_out_sig1 - P_sig1)./P_sig1;

end
```

dir_response.m

```
function [ dir_resp, tau_scan ] = dir_response( look_dir, W, prec, f, x_scan
)
%[ dir_resp, tau_scan ] = dir_response( look_dir, W, prec, f, x_scan )
%   Creates beamplot directional response for an arbitrary array geometry
if length(look_dir) == 1
    look_angle = look_dir.*(pi/180);
else
    look_angle = atan(look_dir(2)/look_dir(1));
end

c = 345;
N = size(prec, 1); % number of microphones

for i=1:N
    d(i) = sqrt( (prec(i,1)-look_dir(1))^2 + (prec(i,2)-look_dir(2))^2 +
(prec(i,3)-look_dir(3))^2);
end

%calc time delay between source and receivers
td = d./c;
% Calc the minimum delay corresponding to the closet microphone to the source
td_min = min(td); % Calc the minimum delay corresponding to the closet
microphone to the source

tau = td - td_min;

for k = 1:length(x_scan)
    for l = 1:N
    d_scan(k,l) = sqrt(sum((prec(l,:) - [x_scan(k) look_dir(2)
look_dir(3)]).^2));
    end
    d_min_scan = min( d_scan(k,:));
    tau_scan(k,:) = (d_scan(k,:) - d_min_scan)./c;
end

for k = 1:length(x_scan)
    for l = 1:N
        dir_resp_W(k,l) = (1./N)*W(l).*(exp(-j*2*pi*f.*(tau_scan(k,l)-
tau(l))));
    end
```

```
end

dir_resp = sum(dir_resp_W, 2);

figure()
plot(x_scan, 10.*log10(abs(dir_resp)))


end
```

DS_beamformer.m

```
function [ ya, z, pout ] = DS_beamformer( x, Fs, look_dir, W, prec )
%[ ya, z, pout ] = DS_beamformer( x, Fs, look_dir, W, prec )
%   x = actual samples at each time step (NxL matrix) N = # of receivers L
%       = # of samples. Each column corresponds to one sample time.
%   Fs = sample rate
%   look_dir = coordinates [x y z] for beamformer to focus on
%   W = channel weighting vector 1xN (# of mics) ex: [1 2 1]
%   prec = 3D coordinate postions of each microphone [x y z]

%   ya = delayed/aligned samples corresponding to beamformer look direction
%   z = summed output of all delayed/aligned samples [1xL vector]
%   pout = output power of z for designated look direction
%

%   generate time series

L = size(x,2); % # of samples L

n = (0:(L-1)); %sample index #

Ts = 1./Fs; %sampling period

t_tx = n.*Ts; % create source time series starting at t = 0

c = 345;
N = size(prec, 1);  %calculate number of receivers

%calc distance between look point and current receiver
for i=1:N
d(i) = sqrt( (prec(i,1)-look_dir(1))^2 + (prec(i,2)-look_dir(2))^2 +
(prec(i,3)-look_dir(3))^2);
end

%calc time delay between source and receivers
td = d./c
% Calc the minimum delay corresponding to the closet microphone to the source
td_min = min(td)

tau = td - td_min; % convert to time delay rewlative to closest mic
```

```
k = round(tau./Ts) % convert relative time delay to equivalent sample delay
k_max  = max(k);

%align samples according to specified BF look direction
for i = 1:N
    ya(i,:) = W(i) .* x(i, ((k(i)+1):(L- k_max + k(i))));
end

%sum all aligned samples to calculate output of beamformer
z = sum(ya)./N;

pout = var(z) % calculate output power of signal

end
```

DS_freq_response.m

```
function [ beam_pwr ] = DS_freq_response( f_sweep, x_scan, Fs, psrc, prec, W
)
%[ beam_pwr ] = DS_freq_response( f_sweep, x_scan, Fs, psrc, prec, W )
%   Detailed explanation goes here

L = length(f_sweep);


 for i = 1:L % generate sinusoid at each frequency of f_sweep
    sine = sine_gen(1, f_sweep(i), Fs, 3);

    % simulate each sinusoid with mic array
    [x, ~, ~] = sim_3D( sine, Fs, psrc, prec, 0);

    %scan all x axis values on a line and look at BF output power
    for j = 1:length(x_scan)                          %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x, Fs, [x_scan(j) psrc(2) psrc(3)],
W, prec);
     beam_pwr(i,j) = pout_scan; %beampower matrix dependant on x position and
frequency
    end
 end

 figure()

 mesh(x_scan, f_sweep, beam_pwr./.5)
 title('Beamformer Frequency Response')
 xlabel('x direction (m)')
 ylabel('frequency (Hz)')
 zlabel('Gain')

 figure()

 title('Broadband Directional Response')
 plot(x_scan, sum(beam_pwr./.5)./length(f_sweep))
```

```
 xlabel('x direction (m)')
 ylabel('Gain')

end
```

DS_lin_dir_response.m

```
function [ P_sig, P_ds ] = DS_lin_dir_response( look_dir, sweep_angles, N, d
,f, var_n )
%Plots 2D directional response beampattern of an equispaced linear array with
%equal weights (DS beamformer)
%   [ P_sig, P_ds ] = lin_dir_response( look_dir, sweep_angles, c, N, d ,f,
var_n )
% look_dir = look direction /steered direciton of beamformer (degrees)
% sweep_angles = directions evaluated for beampattern response 1xL (degrees)
% N = # of receivers
% d = distance between each microphone
% f = signal frequency
% var_n = noise variance for noise field model
% P_sig = signal power for each sweep angle
% P_ds = signal + noise for each sweep angle



c = 345;
k = (2*pi*f/c);
theta = look_dir*(pi/180); % beamformer look direction
psi = sweep_angles*(pi/180); %beamformer directional response sweep angles

% calculate directional signal power from derived formula
P_sig = abs(sin(pi*f*N*d.*(cos(psi)-
cos(theta))./c)./(N*sin(pi*f*d.*(cos(psi)-cos(theta))./c))).^2;

% calculate noise field correlation matrix
for m = 1:N
    for n =1:N
        d_mn = d*abs(m-n);
        R_mn(m,n) = (var_n/(2*pi))* besselj(0,k*d_mn);
    end
end

%Calculate noise power
P_n = (1/N)^2 * sum(sum(R_mn));

%Add signal and noise power together to get total DS beamformer power
P_ds = P_sig + P_n;

figure()
plot(psi.*(180/pi), 10.*log10(P_ds),psi.*(180/pi), 10.*log10(P_sig))
title('Delay Sum Beamformer Directional Response w/ Added Noise')
xlabel('direction (degrees)')
ylabel('Gain (dB)')
legend ('Signal + Correlated Noise','Signal','Location', 'South')
```

```
end
```

DS_metrics.m

```
function [ SNR_out, nr_factor, sd_factor, SNR_in, P_out, P_out_n, P_out_sig1,
P_sig1, z_DS, z_n ] = DS_metrics( src, noise, psrc, prec,FS )
%Calculates output metrics for DS beamformer run on source and noise
%   only signals with known look direction
%
%Run DS Beamformer on src and noise
[ ~, z_DS, pout ] = DS_beamformer( src, FS, psrc, ones(1,9), prec );
[ ~, z_n, pout_n ] = DS_beamformer( noise, FS, psrc, ones(1,9), prec );

%Calculate metrics
    P_in = var(src(1,:));
    P_n = var(noise(1,:));
    P_sig1 = P_in-P_n;
    SNR_in = P_sig1./P_n;

    P_out = pout;
    P_out_n = pout_n;
    P_out_sig1 = pout-pout_n;



    SNR_out = (pout-pout_n)./pout_n;
    nr_factor = var(noise(1,:))./P_out_n;
    sd_factor = abs(P_out_sig1 - P_sig1)./P_sig1;


end
```

DS_metrics_sweep.m

```
%%% DS metrics Sweep

Gain = 1;

% set path where all preprocessed test cases are located
datadir = fullfile('Outdoor_Perimeter_11172011');

% set path where all noise only recording are found
datadir_noise = fullfile('Outdoor_Perimeter_11172011','Noise');

% set path where array measurments are found
datadir_src = fullfile('Outdoor_Perimeter_11172011','Array Measurements');
load(fullfile(datadir_src,'Array_Meas_11_17'))

% Load file containing the list of the names of all of the case files
load(fullfile(datadir,'DataList.mat'))
```

```
for i = 1:length(datalist)
    current_case = datalist(i).name; %extract the name of the current case
    display(['Processing ' current_case])

    load(fullfile(datadir,current_case)) %load the data from the current case

    src = Gain.*TimeDataPP; %amplify signal
    clear TimeDataPP
    clear time

    noise_header = current_case(1:end-8);
    load(fullfile(datadir_noise, [noise_header 'noise_PP.mat'])) %load noise
that corresponds to the current case

    noise = Gain.*TimeDataPP; %amplify noise signal
    clear TimeDataPP
    clear time

    if i == 1
        src_pos = psrc(1,:);
    end
    if i ==6
        src_pos = psrc(2,:);
    end
    if i == 11
        src_pos = psrc(3,:);
    end
    if i == 16
        src_pos = psrc(4,:);
    end
    if i == 21
        src_pos = psrc(5,:);
    end
    if i == 26
        src_pos = psrc(6,:);
    end
    if i == 31
        src_pos = psrc(7,:);
    end

    [ SNR_out, nr_factor, sd_factor, SNR_in, ~, ~, ~, ~, z_DS, z_n ] =
DS_metrics( src', noise', src_pos, prec,FS );

    save(fullfile('DS Beamformer', ['DS_' current_case(1:end-7)]
),'SNR_in','SNR_out','nr_factor','sd_factor','z_DS','z_n','FS','src','noise',
'Gain','current_case','src_pos');
    wavwrite((1./max(abs(z_DS))).*z_DS, fullfile('BF out
wavs',[current_case(1:end-7) '_DSBF']))
    clear SNR_in SNR_out nr_factor sd_factor z_DS z_n current_case src noise
FS
end
```

DS_noise_tests.m

```
%% Noise Analysis for DS Beamformer Script

c = 345; % speed of sound m/s
N = 9; % # of receivers
d = .08; % distance between receivers in linear array
f = 100:4000; %frequency for beamplot Hz
var_n = 1; % noise variance
theta = 90*(pi/180); % beamformer look direction
psi = (0:.99:180)*(pi/180); %beamformer directional response sweep angles

for i = 1:length(f)
k = (2*pi*f(i)/c)

% calculate directional signal power from derived formula
P_sig = abs(sin(pi*f(i)*N*d.*(cos(psi)-
cos(theta))./c)./(N*sin(pi*f(i)*d.*(cos(psi)-cos(theta))./c))).^2;

%calculate noise correlation matrix      samples     var  L
%[ noise_corr, R_corr ] = Spacial_noise_R( 1000, prec, var_n,  10);

for m = 1:N
    for n =1:N
        d_mn = d*abs(m-n);
        R_mn(m,n) = (var_n/(2*pi))* besselj(0,k*d_mn);
    end
end

R_un = var_n.* eye(N);

%Calculate noise power
P_n = (1/N)^2 * sum(sum(R_mn));
P_n_un = (1/N)^2 * sum(sum(R_un));

%Add signal and noise power together to get total DS beamformer power
P_ds(i,:) = P_sig + P_n;
P_ds_un(i,:) = P_sig + P_n_un;

% figure()
% plot(psi.*(180/pi), 10.*log10(P_ds),psi.*(180/pi),
10.*log10(P_sig),psi.*(180/pi), 10.*log10(P_ds_un))
% title('Delay Sum Bemaformer Directional Response w/ Added Noise')
% xlabel('direction (degrees)')
% ylabel('Gain (dB)')
% legend ('Signal + Correlated Noise','Signal', 'Signal + Uncorrelated
Noise','Location', 'South')

end

figure()
 mesh(psi*(180/pi),f,10*log10(P_ds));
 title('new noise model')
 xlabel('Direction (degrees)')
```

```
ylabel('frequency (Hz)')
zlabel('gain (dB)')

figure()
mesh(psi*(180/pi),f,10*log10(P_ds_un));
title('white noise')
xlabel('Direction (degrees)')
ylabel('frequency (Hz)')
zlabel('gain (dB)')
```

DS_Simulation1_noiseBW.m

```
%%% DS Simulation Test 1 %%
%%% varying noise BW to be compared with weiner and ST approach.

prec = [ .956   0   .445;
         0      0   .458;
         -.915  0   .445;
         .956   0   1.368;
         0      0   1.358;
         -.915  0   1.278;
         .956   0   2.139;
         0      0   2.133;
         -.915  0   2.075];

psrc = [0 40 0];
[ src, ~, src_pwr ] = sim_3D( bftest0, FS, psrc, prec );
fc = [ 50 100 200 300 500 800 1000 2000 3000 ];

pout = zeros(1,length(fc));
pout_n = zeros(1,length(fc));
SNR_out = zeros(1,length(fc));
for i = 1:length(fc)
    current_fc = fc(i)
    [ noise_corr, R_vv ] = plane_noise_R_3D( prec, src_pwr, length(src),
fc(i),7,FS );
    x = src+noise_corr;
    [ ~, z, pout ] = DS_beamformer( x, FS, psrc, ones(1,9), prec );
    [ ~, z_n, pout_n ] = DS_beamformer( noise_corr, FS, psrc, ones(1,9), prec
);
    pout_all(i) = pout;
    pout_n_all(i) = pout_n;
    SNR_out(i) = (pout-pout_n)./pout_n;

end
```

DS_Simulation2_B123_noise_bftest0.m

```
%%% DS Simulation Test 2 %%
%%% Simulation With bftest0 src + B123 noise (recorded)

%Inititalize
```

```
FS = 8000;
prec = [ .956   0   .445;
         0      0   .458;
         -.915  0   .445;
         .956   0   1.368;
         0      0   1.358;
         -.915  0   1.278;
         .956   0   2.139;
         0      0   2.133;
         -.915  0   2.075];

psrc = [0 10 0];
[ x1, ~, src_pwr ] = sim_3D( bftest0, FS, psrc, prec );

src = .5.*x1+noise(1:length(x1),:)';

%Run DS Beamformer on src and noise
[ ~, z, pout ] = DS_beamformer( src, FS, psrc, ones(1,9), prec );
[ ~, z_n, pout_n ] = DS_beamformer( noise', FS, psrc, ones(1,9), prec );

%Calculate metrics
    P_out = pout;
    P_out_n = pout_n;
    P_out_sig1 = pout-pout_n;
    P_sig1 = var(src(1,:))-var(noise(:,1));
    SNR_in = (var(src(1,:))-var(noise(:,1)))./var(noise(:,1));

    SNR_out = (pout-pout_n)./pout_n;
    nr_factor = var(noise(:,1))./P_out_n;
    sd_factor = abs(P_out_sig1 - P_sig1)./P_sig1;
```

fractional_downsample.m

```
function yout = fractional_downsample(yin,Fsin,Fsout,tol)
% function yout = fractional_downsample(yin,Fsin,Fsout,tol)
% This function combines decimate and resample to downsample yin from Fsin
% to Fsout. Inputs are:
%     yin: input signal
%     Fsin: sample rate for yin (Hz)
%     Fsout: desired output sample rate (Hz)
%     tol: tolerance for rational approximation to Fsin/Fsout (optional)

    if nargin < 4
        tol = .001;
    end
    [~,ncol] = size(yin);
%    yin = yin(:);
%    ny = length(yin);
    nd = floor(Fsin/Fsout);
    [p,q] = rat(nd*Fsout/Fsin,tol);

    if nd>0
        ydec = decimate(yin(:,1),nd);
    else
```

```
        ydec = yin(:,1);
    end
    yout1 = resample(ydec,p,q);
    nyout = length(yout1);
    if ncol>1
        yout = zeros(nyout,ncol);
        yout(:,1) = yout1;
        for j=2:ncol
            if nd>0
                ydec = decimate(yin(:,j),nd);
            else
                ydec = yin(:,j);
            end
            yout1 = resample(ydec,p,q);
            yout(:,j) = yout1;
        end
    else
        yout = yout1;
    end

end
```

Init_3D_SIM.m

```
%% Initialize Sourcnce and receiver locations
clear variables
%%
psrc = [0 40 0];
pnoise = [0 10 1];
prec = [ .956   0   .445;
         0      0   .458;
         -.915  0   .445;
         .956   0   1.368;
         0      0   1.358;
         -.915  0   1.278;
         .956   0   2.139;
         0      0   2.133;
         -.915  0   2.075];
W = [1 1 1 1 1 1 1 1 1];
x_scan = linspace(-20,20,300);


 %% load sounds
 load('handel');
 handel = y;
 bftest0 = wavread('bftest0');
 bftest1 = wavread('bftest1');
 sine = sine_gen(1, 2000, 40000, 1);                          %var %gausian
width
 [ noise_corr, R_corr ] = Spacial_noise_R( length(x1), prec, .1,  1.5);

 %% Execute Simulation

 %% Single source
```

```
 [ x1, ~, src_pwr ] = sim_3D( bftest0, Fs, psrc, prec );

 %% Dual Sources
 %src 1
 [ x1, ~, src_pwr ] = sim_3D( bftest0, Fs, psrc, prec );

 %src2
 [ x2, ~, src_pwr2 ] = sim_3D( handel, Fs, pnoise, prec );

 min_L = min([size(x1,2) size(x2,2)]);

 x1 = x1(:,1:min_L);
 x2 = x2(:,1:min_L);
 x = x1+x2;

 %% Plot
 x_scan = linspace(-10,10,300);               % y_plane z_plane weighting
vector
 [beam_pwr] = x_beam_plot( noise, Fs, prec, x_scan, psrc(2), psrc(3), W);

 %% Run DS beamformer focused on each source

 [ ~, z_DS, ~ ] = DS_beamformer( x, Fs, psrc, W, prec );

 [ ~, z_SNR, ~ ] = DS_beamformer( x, Fs, psrc2, h_max', prec );

 sound(x(1,:))
 sound(z_DS)
 sound(z_SNR)

 %% Run Freq Resp
 f_sweep = linspace(80, 3000, 20);
 x_scan = linspace(-10, 10, 300);
 [ beam_pwr ] = DS_freq_response( f_sweep, x_scan, Fs, psrc, prec, h_max');
```

Init_SIM_script.m

```
%% Load Hallelujah
load handel

%% Initialize Source Location
psrc = [0 20]

%% Initialize Receiver Locations
p1= [-2.5 0]
p2= [-1.5 0]
p3 = [-.5 0]
p4 = [.5 0]
p5 = [1.5 0]
p6 = [2.5 0]

%% Run Simulator
```

```
xy_sim_6( y, Fs, psrc, p1, p2, p3, p4, p5, p6 );
```

lin_dir_response.m

```
function [ P_bf, z ] = lin_dir_response( look_dir, sweep_angles, f, W, N  )
%Computes and plots the 2D directional repsonse of a linear array with equal
%spacing amnd user defined channel weightings W.
%
%NOTE: some code used courtesy of Brian D. Jeffs
% Associate Professor
% Dept. of Electrical and Computer Engineering
% Brigham Young University
% March 2008
%
%[ output_args ] = lin_dir_response( look_dir, sweep_angles, f, W, N  )
%
% N =  % no. of array elements
% f = frequency (Hz)
c = 345;

d = c/f/2; % element spacing
look_dir = look_dir*(pi/180);
d_s = exp(j*2*pi*f*d/c*cos(look_dir)*[0:N-1]).';
R_s = d_s*d_s';

% compute steering vector samples for beam resp. plot
psi = sweep_angles.'*pi/180;
D_b = exp(j*2*pi*f*d/c*cos(psi)*[0:N-1]).';

% conventional windowed beamformer case
h = d_s.*W;
z = h'*D_b./sum(abs(W)); % beam response after weighting
P_bf = abs(z).^2;

figure()
plot(sweep_angles,10*log10(P_bf))
title( 'Directional Response for Weighted Beamformer')
xlabel('direction (degrees)')
ylabel( 'Gain (dB)')
end
```

max_SNR.m

```
function [h_max, SNR_max, SNR_DS, SNR_mSNR, z_src ] = max_SNR( psrc, prec,
src, noise_corr, Fs, x_scan)
% %[h_max, SNR_max, SNR_DS, SNR_mSNR ] = max_SNR( psrc, prec, src,
noise_corr, Fs, x_scan)
%     psrc = source position [x y z]
%     prec = receiver positions Nx3 [x y z]
%     src = source sound 1xL
%     noise_corr = NxL matrix of spatially correlated noise located at x = 0
%     Fs = sampling rate
%     x_scan = x position samples for output plots
```

```matlab
%
%     h_max = Nx1 array of calculated weights for each receiver channel
%     SNR_max = eigenvalue corresponding to h_max eigen vector
%     SNR_DS = signal to noise ratio of unity weighted DS beamformer at
%       source look direction
%     SNR_mSNR = signal to noise ratio of Max SNR algorithm using h_max as
%       DS weights

N = size(prec,1);

a (1:N) = 1;
aat = a'*a;
W(1:N) = 1;

%src simulation
[ x1, ~, ~ ] = sim_3D( src, Fs, psrc, prec );

%[ noise, ~, ~ ] = DS_beamformer( noise_corr, Fs, -1.*pnoise, W, prec );

%superposition source and noise signals
%force matrices to be same length
min_L = min([size(x1,2) size(noise_corr,2)]);

x1 = x1(:,1:min_L);
x2 = noise_corr(:,1:min_L);

x = x1+x2;

% align/delay noise source samples in direction of the source
[ va, ~, ~ ] = DS_beamformer( noise_corr, Fs, psrc, W, prec );

%STAT CALCS
% Find Eigenvector that corresponds to max eigenvalue of constraint eqn
R_vv = corr( va');
R_vv_inv = inv(R_vv);
A = var(src).*R_vv_inv*aat; %Matrix for eigenvalue calc.... A*x = SNR*x
[h_max, SNR_max] = eigs(A,1)

% Evaluate DS response
%calculate received power from the source as a function of position
for i = 1:length(x_scan)                            %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x1, Fs, [x_scan(i) psrc(2)
psrc(3)], W, prec );
     beam_pwr_sDS(i) = pout_scan;
end

%calculate received power from the noise as a function of position
for i = 1:length(x_scan)                            %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x2, Fs, [x_scan(i) psrc(2)
psrc(3)], W, prec );
     beam_pwr_nDS(i) = pout_scan;
end

%calculate total received power from at the output of the beamformer
```

```
for i = 1:length(x_scan)                            %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x, Fs, [x_scan(i) psrc(2) psrc(3)],
W, prec );
     beam_pwr_DS(i) = pout_scan;
end

% Calculate DS SNR
[ ~, ~, pout_sDS ] = DS_beamformer( x1, Fs, psrc, W, prec );
[ ~, ~, pout_nDS ] = DS_beamformer( x2, Fs, psrc, W, prec );

SNR_DS = 10.*log10(pout_sDS./pout_nDS);

% Evaluate Max SNR response

for i = 1:length(x_scan)                            %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x1, Fs, [x_scan(i) psrc(2)
psrc(3)], h_max', prec );
     beam_pwr_sSNR(i) = pout_scan;
end

for i = 1:length(x_scan)                            %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x2, Fs, [x_scan(i) psrc(2)
psrc(3)], h_max', prec );
     beam_pwr_nSNR(i) = pout_scan;
end

for i = 1:length(x_scan)                            %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x, Fs, [x_scan(i) psrc(2) psrc(3)],
h_max', prec );
     beam_pwr_SNR(i) = pout_scan;
end

%Calc actual Max SNR SNR
[ ~, z_src, pout_sSNR ] = DS_beamformer( x1, Fs, psrc, h_max', prec );
[ ~, z_noise, pout_nSNR ] = DS_beamformer( x2, Fs, psrc, h_max', prec );

SNR_mSNR = 10.*log10(pout_sSNR./pout_nSNR);

% Plot DS response

figure()
plot(x_scan, beam_pwr_sDS, x_scan, beam_pwr_nDS, 'r')
title('DS Source and Noise directional response')
xlabel('x location (m)')
ylabel('Beam power (W)')

% figure()
% plot(x_scan, beam_pwr_DS)
% title('DS total output directional response')
% xlabel('x location (m)')
% ylabel('Beam power (W)')

%Plot Max SNR response
```

```
figure()
plot(x_scan, beam_pwr_sSNR, x_scan, beam_pwr_nSNR, 'r')
title('Max SNR Source and Noise directional response')
xlabel('x location (m)')
ylabel('Beam power (W)')


% figure()
% plot(x_scan, beam_pwr_SNR)
% title('Max SNR total output directional response')
% xlabel('x location (m)')
% ylabel('Beam power (W)')

% figure()
% plot(x_scan, (beam_pwr_nSNR./beam_pwr_nDS))
% title('Noise Gain Directional Response')
% xlabel('x position (m)')
% ylabel('noise gain')

figure()
subplot(2,1,1)
plot(0:1/Fs:(length(src)-1)*(1/Fs),src)
title('Original Source Signal')
xlabel('time (s)')
ylabel('Amplitude')

subplot(2,1,2)
plot(0:1/Fs:(length(z_src)-1)*(1/Fs), z_src)
title('Source Signal After Beamforming')
xlabel('time (s)')
ylabel('Amplitude')

end
```

plane_noise_R.m

```
function [ R_vv ] = plane_noise_R( prec, look_dir, var_n, f )
%Builds a NxN correlation matrix for random directional plane wave noise
%model.
%   prec = receiver postions [x1 y1 z1; ... xN yN zN]
%   look_dir = point/ direction that the beamformer is looking [x y z]
%   var_n = noise variance scale factor for correlation coeficients
%   f = operating frequency of  beamformer

% Calculate constants
N = size(prec,1); % number of mics
c = 345; %speed of sound
k = 2*pi*f/c; %wave number

%calculate distances between receivers and put into NxN matrix
for m = 1:N
    for n = 1:N
        d_mn(m,n) = sqrt( sum((prec(m,:) - prec(n,:)).^2) );
    end
end
```

```
%calculate distances from each mic to look direction [x y z]
for i = 1:N
    d(i) = sqrt( sum((look_dir - prec(i,:)).^2));
end

%convert distances to each mic into relative time delays for each mic
td = (d-min(d))./c;

% calculate noise field correlation matrix
for m = 1:N
    for n =1:N
        R_vv(m,n) = var_n* besselj(0,k.*d_mn(m,n)).*cos(2*pi*f*abs((td(n)-
td(m))));
    end
end

end
```

plane_noise_R_3D.m

```
function [ noise_corr, R_vv ] = plane_noise_R_3D( prec, var_n, length,
fc,order,Fs )
%Creates spatially correlated low pass filtered (correlated in time) noise
%   prec = receiver postions [x1 y1 z1; ... xN yN zN]
%   var_n = variance of the noise
%   length = # oof samples of the created noise
%   fc = upper cutoff frequency of the desired noise (Hz)

% Calculate constants
N = size(prec,1); % number of mics
c = 345; %speed of sound
kc = 2*pi*fc/c; %wave number of upper cutoff frequency

%Correlate random white noise in time by low pass filtering
[B_lp,A_lp] = butter(order, (2/Fs)*fc);

noise =  randn(N,length);
noise_lp = (filtfilt(B_lp, A_lp ,noise'))';


%calculate distances between receivers and put into NxN matrix
d_mn = zeros(N,N);
for m = 1:N
    for n = 1:N
        d_mn(m,n) = sqrt( sum((prec(m,:) - prec(n,:)).^2) );
    end
end

% calculate noise field correlation matrix
R_vv = zeros(N,N);
for m = 1:N
    for n =1:N
```

```
        if d_mn(m,n) == 0
            R_vv(m,n) = 1;
        else
            R_vv(m,n) = sinint(kc.*d_mn(m,n))./(kc*d_mn(m,n));
        end
    end
end

% Spatially Correlate the noise using R_vv
norm = mean(var(noise_lp'));
noise_corr = sqrt((1/norm)).*sqrt(var_n).*sqrtm(R_vv)*noise_lp;
end
```

Preprocess_data.m

```
% Preprocess_data_11172011
% Preprocessing script for array data for outdoor 11/17/2011 data collect
datadir = fullfile('Outdoor_Perimeter_11172011','Noise');
load(fullfile(datadir,'DataList.mat'))
ndata = length(datalist);
for j=1:ndata
    datafile = datalist(j).name;
    disp(['Processing ' datafile])
    Array_preprocess(datadir,datafile,8000,150);
end
clear j datafile ndata
```

receiver.m

```
function [ t_tx, t_rx, snd_rx ] = receiver( snd_tx, Fs, d, c, a)
%[ t_tx, t_rx, snd_rx ] = receiver( snd_tx, Fs, d, c)
%   sound = amplitude samples of sound file from wave (-1 : 1)
%   Fs = sampling rate of source sound (samples/second)
%   d = distance between source and receiver (m)
%   c = speed of sound wave (345 m/s)
%   a = attenuation facor 0:1
%   t_tx = transmitted time series (s)
%   t_rx = received signal time series
%   td = time delay from source to receiver

%   generate time series

N = length(snd_tx); % of samples N

n = (0:(N-1)); %sample index #

Ts = 1./Fs %sampling period

t_tx = n.*Ts;

%   aquire received signal
```

```
td = d./c; %time delay from source to receiver

t_rx = t_tx + td %calc received signal time series

snd_rx = a.*snd_tx; % account for attenuation

figure (1)
title('source to receiver time delay')

subplot(2,1,1)
plot (t_tx, snd_tx)
xlabel('time [sec]')
ylabel('Amplitude')
title('Source Signal')

subplot(2,1,2)
plot(t_rx, snd_rx)
xlabel('time [sec]')
ylabel('Amplitude')
title('received signal')

end
```

rect_polar.m

```
function [ xy ] = rect_polar( r, angle_sweep )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

theta = angle_sweep*(pi/180);

x = r*cos(theta);
y = r*sin(theta);

xy = [x; y]';

end
```

semilogx_spectrum.m

```
function [ f, Y_f ] = semilogx_spectrum( y_t,FS )
%Plots the Spectrum of y_t with same number of points as y_t
%   Detailed explanation goes here

f = FS*(0:1./(length(y_t)):1-(1./length(y_t)));
Y_f = abs(fft(y_t))./length(y_t);
semilogx(f,Y_f);
end
```

sim_3D.m

```
function [ x, ya_sim, src_pwr ] = sim_3D( snd_tx, Fs, psrc, prec, dist_atten)
%[ x, ya_sim, src_pwr ] = sim_3D( snd_tx, Fs, psrc, prec, dist_atten)
%  Calculates outputs of a microphone array with one source in 3D
%   snd_tx = samples of source signal (Lx1)
%   Fs = Sampling rate or sound source
%   psrc = source position [x y z]
%   prec = microphone positions Nx3 matrix: row = source #, columns = x y z
%   dist_atten = accounts for 1/r attenuation due to source to array
%       distance. Leave empty if no attenuation is desired. Enter '1' if
%       attenuation is desired
%
%   x = received sound samples at each corresponding mic (each row is a
different receiver
%       each column corresponds to sample times)
%   ya_sim = simulated aligned output values after added noise (if any)
%   src_pwr = calculates the power of the source signal
if nargin == 4
    dist_atten = 0;
end

%   generate time series

L = length(snd_tx); % # of samples in source signal

n = (0:(L-1)); %sample index #

Ts = 1/Fs; %sampling period

t_tx = n.*Ts; % create source time series starting at t = 0

%Calculate source power
src_pwr = var(snd_tx);

%Calc source -> receiver distances
c = 345;
N = size(prec, 1); % number of microphones

for i=1:N
    d(i) = sqrt( (prec(i,1)-psrc(1))^2 + (prec(i,2)-psrc(2))^2 + (prec(i,3)-
psrc(3))^2);
end

if(min(d) < 1)
    error('Source cannot be closer than one meter to the array!')
end

% Calculate attenuation factor
a = 1./d;
if(~dist_atten)
a(:) = 1;
end
```

```
%calc time delay between source and receivers
td = d./c;

% Calc the minimum delay corresponding to the closet microphone to the source
td_min = min(td); % Calc the minimum delay corresponding to the closet
microphone to the source

%calculate time series for each receiver where t = 0 is when the source
signal first
% hits the closest mic to the source
for i = 1:N
    t(i,:) = td(i) + t_tx - td_min;
end

%shift data to correspond with time indicies
%data starts when source signal first hits the closest microphone
%removes preceding '0's due to initial source to receiver delay
for i = 1:N
    x(i, round((t(i,:)/Ts) +1)) = a(i).*snd_tx;
end

K = length(x);
t_rec = (0:Ts:(K-1)*Ts);

%add noise here
% y = x + n
y = x;

%align array of signals corresponding to the source look direction
for i = 1:N
    ya_sim(i,:) = y(i,round((t(i,:)/Ts) +1));
end

%Plot locations of source and receivers
% figure (1)
%
% hold on
% grid on
%
% title('Source and Receiver Loacations')
%
% s = scatter3(psrc(1),psrc(2), psrc(3), 'g', 'filled');
% r = scatter3(prec(:,1),prec(:,2),prec(:,3), 'b', 'filled'); view(30,20);
%
% xlabel('x-direction (m)')
% ylabel('y-direction (m)')
% zlabel('z-direction (m)')
% legend([s r], 'Source', 'Receivers')
% hold off
%
% %plot received signals for each mic
% figure (2)
% hold on
% title('Received Signals')
%
```

```
% for i = 1:N
% ax(i) = subplot(N,1,i);
% plot(t_rec,x(i,:))
% label = ['Receiver ' num2str(i)];
% ylabel(label)
% end
%
% linkaxes(ax, 'xy')
% hold off
 end
```

sim_dir_resp.m

```
function [ P_out,P_sig, P_out_n, P_in_n, src_pwr ] = sim_dir_resp( src, Fs,
look_dir, prec, W, angle_sweep, var_n, f )
%Plots the actual simulated directional resopnse of an input signal
%(preferreably a sinusoid) by varying the actual sourec location while
%keeping the beamformer looking in a specified direction.
%   src = source signal of any length
%   Fs = sampling rate of source signal (samples/sec)
%   look_dir = beamformer look point [x y z] beamformer focuses on
%   prec = postion of receivers [x1 y1 z1; ...xN yN zN]
%   W = weighting of each channel. Must be length N
%   angle_sweep = direction points (degrees) to be tested and plotted for the
directcional
%       response
%
%   P_out = output power array for every x_scan point
%   P_sig = signal power at output of beamformer
%   P_out_n = noise pwer at the output of the beamformer
%   P_in_n = input noise power at the receivers
%   src_power = orignal source power to be compared with the output of the
%       beamformer

%Calculate number of sensors in array
N = size(prec,1);

%Calculates distance of look point to origin
r = sqrt(look_dir(1).^2 + look_dir(2).^2);

%Converts sweep angles to rectangular coordinates for simulator
xy = rect_polar(r, angle_sweep);

%Calculate noise correlation matrix based on time delays and receiver
%positions and frequency
[ R_vv ] = plane_noise_R( prec, look_dir, var_n, f );


for i = 1:length(angle_sweep)
    %simulate source from each angle in angle sweep
    [ x, ~, src_pwr ] = sim_3D( src, Fs, [xy(i,1) xy(i,2) look_dir(3)],
prec);

    %Beamform source signal for designed beamformer look direction (not
```

```
    %actual source location)
    [ ya, z_sig, P_sig(i) ] = DS_beamformer( x, Fs, look_dir, W, prec );

    %generate random noise corresponding to R_vv correlation matrix
    L = size(ya,2);
    va = sqrtm(R_vv)*randn(N,L);

    %Weight and sum noise with same weights used to beamform the signal
    %NOTE: noise does not need to be delayed due to the correlation matrix
    %taking that into account
    z_n = (W'*va)./N;

    %Superposition source and noise signals after beamforming
    z = z_sig + z_n;

    % Calculate input noise power
    P_in_n(i) = mean(var(va'));
    %Caluclate output noise power after weighting
    P_out_n(i) = var(z_n);
    %Calculate total output power of noise and signal combined
    P_out(i) = var(z);
end

figure()
plot(angle_sweep, 10.*log10(P_out./src_pwr),'b',
angle_sweep,10.*log10(P_sig./src_pwr),'r',
angle_sweep,10.*log10(P_out_n./P_in_n),'g' )
title('Simulated Directional Response')
xlabel('direction (degrees)')
ylabel('Gain (dB)')
end
```

sim_steered_resp.m

```
function [ P_sig, P_out_n, P_in_n, P_out, src_pwr] = sim_steered_resp( src,
Fs, psrc, angle_sweep, prec, var_n, f, max_snr_weights )
%Sweeps beamformer look direction at each angle of angle sweep and
%calculates the signal, noise and total power at the output of the
%beamformer.
%   src = source signal of any length
%   Fs = sampling rate of source signal (samples/sec)
%   look_dir = beamformer look point [x y z] beamformer focuses on
%   prec = postion of receivers [x1 y1 z1; ...xN yN zN]
%   angle_sweep = direction points (degrees) to be tested and plotted for the
directional
%       response
%   max_snr_weights-> if = to'1' then max_snr weights will be calculated
%   and applied. If not = '1', regular DS weights will be used (1/N)
%
%   P_out = output power array for every angle_sweep point
%   P_sig = signal power at output of beamformer
%   P_out_n = noise power at the output of the beamformer
%   P_in_n = input noise power at the receivers
%   src_power = orignal source power to be compared with the output of the
```

```
%       beamformer

%Calculate number of sensors in array
N = size(prec,1);

%Calculates distance of look point to origin
r = sqrt(psrc(1).^2 + psrc(2).^2);

%Converts sweep angles to rectangular coordinates for simulator
xy = rect_polar(r, angle_sweep);

%simulate source at actual source location
[ x, ~, src_pwr ] = sim_3D( src, Fs, psrc, prec);

for i = 1:length(angle_sweep)
    %Calculate noise correlation matrix based on time delays (look angle) and
receiver
    %positions and frequency
    [ R_vv ] = plane_noise_R( prec, [xy(i,1) xy(i,2) psrc(3)], var_n, f );
    R_vv_sum(i) = sum(sum(R_vv));

    %Beamform source signal at every look direction in angle sweep
    [ ya, ~, ~ ] = DS_beamformer( x, Fs, [xy(i,1) xy(i,2) psrc(3)],
ones(N,1), prec );
    L = size(ya,2);

    %generate random noise corresponding to R_vv correlation matrix
    va = sqrtm(R_vv)*randn(N,L);

    %Calculate maximum SNR algorithm coeffiecients
    if max_snr_weights == 1 && var_n ~= 0
    [h_max, ~] = eigs(src_pwr.*inv(R_vv)*ones(N),1);
    W = N.*h_max./sum(h_max);

    else W = ones(N,1); % Use delays sum weights
    end

    %Weight and sum signal and noise with same weights
    %NOTE: noise does not need to be delayed due to the correlation matrix
    %taking that into account
    z_sig = (W'*ya)./N;
    z_n = (W'*va)./N;

    %Superposition source and noise signals after beamforming
    z = z_sig + z_n;

    %Calculate output signal power
    P_sig(i) = var(z_sig);
    % Calculate input noise power
    P_in_n(i) = mean(var(va'));
    %Caluclate output noise power after weighting
    P_out_n(i) = var(z_n);
    %Calculate total output power of noise and signal combined
    P_out(i) = var(z);
```

```
end

figure()
plot(angle_sweep, 10.*log10(P_out./src_pwr),'b',
angle_sweep,10.*log10(P_sig./src_pwr),'r',
angle_sweep,10.*log10(P_out_n./P_in_n),'g' )
title('Simulated Steered Response')
xlabel('Beamformer Look Direction (degrees)')
ylabel('Gain (dB)')
end
```

sine_gen.m

```
function [ sine ] = sine_gen( amp, freq, Fs, length_t )
%[ sine ] = sine_gen( amp, freq, Fs, length_t )
%   Detailed explanation goes here

t = 0:1/Fs:length_t;
sine = amp.*sin(2.* pi.*freq.*t);

end
```

Single_Mic_Wavwrite.m

```
%%% Single Mic wav file extractor

Gain = 1;

% set path where all preprocessed test cases are located
datadir = fullfile('Outdoor_Perimeter_11172011');

% Load file containing the list of the names of all of the case files
load(fullfile(datadir,'DataList.mat'))

for i = 1:length(datalist)
    current_case = datalist(i).name; %extract the name of the current case
    display(['Processing ' current_case])

    load(fullfile(datadir,current_case)) %load the data from the current case

    src = Gain.*TimeDataPP(:,1); %amplify signal
    clear TimeDataPP
    clear time

    wavwrite((1./max(abs(src))).*src, fullfile('BF out wavs',
current_case(1:end-7)))
    clear current_case src FS
end
```

sinint.m

```
function y = sinint(x)
% function y = sinint(x)
% This function evaluates the sine integral function using the identity
%  Si(x) = (1/2*i) (expint(i*x)-expint(-i*x)) + pi/2
% For x<0.001, use polynomial approximation Si(x)~x - x^3/18

    indr = find(abs(x)>=.001);
    inds = find(abs(x)<.001);
    y = zeros(size(x));
    c = sqrt(1/18);
    if ~isempty(indr)
        y(indr) = pi/2+ (expint(1i*x(indr))-expint(-1i*x(indr)))/(2*1i);
    end
    if ~isempty(inds)
        y(inds) = x(inds).*(1-c*x(inds)).*(1+c*x(inds));
    end

end
```

Spacial_noise_R.m

```
function [ noise_corr, R_corr ] = Spacial_noise_R( n_samples, prec, n_var,
L)
%[ noise_corr, R_corr ] = Spacial_noise_R( n_samples, prec, n_var,  L)
%   calculates N channels of spatially correlated noise according to:
%   R_corr(i,j) = exp(-mag(prec(i)-prec(j))^2/L^2)
%
%   n_samples = # of columns of spatially correlated noise
%   prec =receiver positions N x3 [x y z]
%   n_var = nosie variance
%   L = gaussian decay factor

N = size(prec,1);

for i = 1:N
    for j = 1:N
        d_rec = sqrt(sum((prec(i,:)-prec(j,:)).^2));
        R_corr(i,j) = exp(-(d_rec.^2)./(L.^2));
    end
end

noise_corr = sqrt(n_var).*sqrtm(R_corr)*randn(N, n_samples);

end
```

Spatio_Temporal_Filter.m

```
function [ z_ST, W_o,H_ST, R_vv, SSNR_in ] = Spatio_Temporal_Filter( y,
v_only,L,overlap )
%Applies a multichannel Spatio-Temporal Prediction algorithm to a
%multichannel signal given signal+noise samples and noise only samples.
```

```
%   y = signal +noise array (N channels x ..)
%   v_only = noise only array (N channels x...)
%   L = frame size of filter;
%   overlap = number of sample overlap between frames

%Verify that the sample overlap does not exceed block length
if(overlap>L)
    error('Sample block "L" must be larger than sample overlap')
end
N = size(y,1); % Calculate number of mics in array

%Initialize
R_vv = zeros(N*L,N*L); % Initialize noise correlation matrix NLxNL
R_yy = zeros(N*L,N*L); % Initialize signal + noise correlation matrix
m = 0;%intialize noise correlation matrix calculation counter for averaging
later
n = 0;% intialize signal+noise correlation matrix calculation counter for
averaging later
P_in_n_SEG = 0; % initialize average noise power of each frame
P_in_SEG = 0;


%Calculate statistics for each block
for i = 1:L-overlap:max(length(v_only),length(y)) %initialize index to start
at beginning of each block of length L taking sample overlap into account

        % Calculate noise statistics
    if(L+i-1 <= length(v_only)) %if current block will exceed length of input
array, break for loop

        v = v_only(:,i:L+i-1)'; % Take a block of L samples at starting at
current index i

        %Organize NxL matrix containing sample blocks into NLx1 matrix

        v_L = v(:);

        %Calculate current correlation matrix for N blocks of L samples
        m = m+1; % number of times correlation matrix is calculated

        %Average correlation matrix over time
        R_vv = ((m-1)/m)*R_vv + (v_L * v_L')./m;
        %R_vv = lambda*R_vv + (v_L * v_L').*(1-lambda);
        P_in_n_SEG = ((m-1)/m)*P_in_n_SEG + mean(var(v))./m;
    end

    % Calculate source + noise statistics
    if(L+i-1 <= length(y))

        y_L = y(:,i:L+i-1)'; % Take a block of L samples at starting at
current index i

        %Organize NxL matrix containing sample blocks into NLx1 matrix
```

```
        y_k = y_L(:);

        n = n+1;

        R_yy = ((n-1)/n)*R_yy + (y_k*y_k')./n;
        %R_yy = lambda*R_yy + (y_k*y_k').*(1-lambda);
        P_in_SEG = ((n-1)/n)*P_in_SEG + mean(var(y_L))./n;
    end
end

SSNR_in = (P_in_SEG-P_in_n_SEG)./P_in_n_SEG;



%Calculate the optimal spatio-temporal prediction matrix W_o

W_o = ((R_yy(:,1:L)-R_vv(:,1:L))/(R_yy(1:L,1:L)-R_vv(1:L,1:L)))';

H_ST = inv(W_o*inv(R_vv)*W_o')*W_o*inv(R_vv);

for i = 1:L-overlap:length(y)
    if(L+i-1 > length(y)) %if current block will exceed length of input array
data, break for loop
        break
    end
    y_L = y(:,i:L+i-1)'; % Take a block of L samples at starting at current
index i

    %Organize NxL matrix containing sample blocks into NLx1 matrix

    y_k = y_L(:);

    %perform filtering operation using matrix multiply
    z_ST(i:i+L-1,1) = H_ST*y_k;

end

end
```

ST_Test1_filt_length_0_overlap.m

```
%% ST Filter Test 3- Varying filter lengths with 0 overlap

L = [2     3     4     5     6     7     8    10    16    20    24    28
30    32];
overlap = L -L;
z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
z_ST_all = zeros(max(length(noise),length(src)),length(L));
for i = 1:length(L)
    L_current = L(i)
    [ z_ST, ~,H_ST, ~ ] = Spatio_Temporal_Filter( src', noise', L(i),
overlap(i)  );
```

```
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src', noise', L(i),overlap(i),
H_ST, z_ST );

    SNR_in_all(i) = SNR_in;
    SNR_out_all(i) = SNR_out;
    nr_factor_all(i) = nr_factor;
    sd_factor_all(i) = sd_factor;
    P_out_all(i) = P_out;
    P_out_n_all(i) = P_out_n;
    P_sig1_all(i) = P_sig1;
    P_out_sig1_all(i) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_ST_all(1:length(z_ST),i) = z_ST;
end
```

ST_Test2_overlap.m

```
%% ST Filter Test 2- Varying sample overlaps with L =  10 filter
tic;
L(1:10) = 10;
overlap = 0:9;
z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
z_ST_all = zeros(max(length(noise),length(src)),length(L));
for i = 1:length(L)
    current_overlap = overlap(i)
    [ z_ST, ~,H_ST, ~ ] = Spatio_Temporal_Filter( src', noise', L(i),
overlap(i)  );
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src', noise', L(i),overlap(i),
H_ST, z_ST );

    SNR_in_all(i) = SNR_in;
    SNR_out_all(i) = SNR_out;
    nr_factor_all(i) = nr_factor;
    sd_factor_all(i) = sd_factor;
    P_out_all(i) = P_out;
    P_out_n_all(i) = P_out_n;
    P_sig1_all(i) = P_sig1;
    P_out_sig1_all(i) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_ST_all(1:length(z_ST),i) = z_ST;
end
toc;
```

ST_Test3_filt_length_max_overlap.m

```
%% ST Filter Test 3- Varying filter lengths with max overlap
tic
L = [2     3     4     5     6     7     8    10    16    20    24    28
30    32];
%L = [40 70 100 ]; % extended version
overlap = L-1;
```

```
z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
z_ST_all = zeros(max(length(noise),length(src)),length(L));
for i = 1:length(L)
    L_current = L(i)
    [ z_ST, ~,H_ST, ~, SSNR_in] = Spatio_Temporal_Filter( src', noise', L(i),
overlap(i)  );
    [ z_v_filt, SNR_in, SNR_out, SSNR_out, nr_factor, sd_factor, P_out,
P_out_n, P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src', noise',
L(i),overlap(i), H_ST, z_ST );

    SSNR_in_all(i) = SSNR_in;
    SSNR_out_all(i) = SSNR_out_all;
    SNR_in_all(i) = SNR_in;
    SNR_out_all(i) = SNR_out;
    nr_factor_all(i) = nr_factor;
    sd_factor_all(i) = sd_factor;
    P_out_all(i) = P_out;
    P_out_n_all(i) = P_out_n;
    P_sig1_all(i) = P_sig1;
    P_out_sig1_all(i) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_ST_all(1:length(z_ST),i) = z_ST;
end
toc
```

STP_Test4_N.m

```
%% STP Filter Test 4- Varying Mic #s
tic
N = 1:9;
L = [2 3 4 5 6 7 8 10 16 20 24 28 32] ;
overlap = L-1;
z_v_filt_all = zeros(max(length(noise),length(src)),length(N));
z_ST_all = zeros(max(length(noise),length(src)),length(N));
for i = 1:length(N)
    current_receiver_num = N(i)

    for j= 1:length(L)
    [ z_ST, ~,H_ST, ~ ] = Spatio_Temporal_Filter( src(:,1:N(i))',
noise(:,1:N(i))', L(j), overlap(j)  );
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src(:,1:N(i))', noise(:,1:N(i))',
L(j),overlap(j), H_ST, z_ST );

    SNR_in_all(i,j) = SNR_in;
    SNR_out_all(i,j) = SNR_out;
    nr_factor_all(i,j) = nr_factor;
    sd_factor_all(i,j) = sd_factor;
    P_out_all(i,j) = P_out;
    P_out_n_all(i,j) = P_out_n;
    P_sig1_all(i,j) = P_sig1;
    P_out_sig1_all(i,j) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_ST_all(1:length(z_ST),i) = z_ST;
    end
```

```
end
toc
```

Test3_case_sweep.m

```
%%% Algorithm Test 3 Case script
% Opens and executes each case from the 11/17/2011 outdoor data collect
% using the Spatio Temporal Predictions and Wiener Filters.
%
%Set Up:
% 1. Preprocess all cases using "Array_Preprocess" function
% 2. Put all preprocessed cases in their own folder "datadir"(change code to
match)
% 3. Put all noise only files in a separate subfolder "datadir_noise" (change
code to match)
% 4. Create "DataList.mat" file using datalist = dir (make sure only case
%    files are in the current folder)
% 5. Save "DataList.mat" in the folder with all of the cases to be tested
% 6. Make sure all noise file are in this format: 10ft_noise.mat
% 7. Make sure all case files are in this format: 10ft_B.mat
% 8. Verify location wher output wavs will be saved

L = [ 2 6 10 20 30 40 70]; % set filter lengths to be used in sweeps
overlap = L-1;
Gain = 1000; % Set signal gain before filtering

% set path where all preprocessed test cases are located
datadir = fullfile('Array_data','Outdoor_Perimeter_11172011'); %%% <----- 2

% set path where all noise only recording are found
datadir_noise = fullfile('Array_data','Outdoor_Perimeter_11172011','Noise');
%%% <----- 3

% Load file containing the list of the names of all of the case files
load(fullfile(datadir,'DataList.mat'))


for i = 1:length(datalist)

    current_case = datalist(i).name; %extract the name of the current case
    display(['Processing ' current_case])

    load(fullfile(datadir,current_case)) %load the data from the current case

    src = Gain.*TimeDataPP; %amplify signal
    clear TimeDataPP
    clear time

    noise_header = current_case(1:end-8);
    load(fullfile(datadir_noise, [noise_header 'noise_PP.mat'])) %load noise
that corresponds to the current case

    noise = Gain.*TimeDataPP; %amplify noise signal
```

```
    clear TimeDataPP
    clear time

    %Run STP Sweep
    z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
    z_ST_all = zeros(max(length(noise),length(src)),length(L));
    for j = 1:length(L)
        L_current = L(j)
        [ z_ST, ~,H_ST, ~, SSNR_in] = Spatio_Temporal_Filter( src', noise',
L(j), overlap(j)  );
        [ z_v_filt, SNR_in, SNR_out, SSNR_out, nr_factor, sd_factor, ~, ~, ~,
~ ] = BB_Filter_Metrics( src', noise', L(j),overlap(j), H_ST, z_ST );

        SSNR_in_all(j) = SSNR_in;
        SSNR_out_all(j) = SSNR_out;
        SNR_in_all(j) = SNR_in;
        SNR_out_all(j) = SNR_out;
        nr_factor_all(j) = nr_factor;
        sd_factor_all(j) = sd_factor;
        z_v_filt_all(1:length(z_v_filt),j) = z_v_filt;
        z_ST_all(1:length(z_ST),j) = z_ST;
    end

    %save output metrics to appropriate Test 3 folder
    save(fullfile('Spatio-Temporal Prediction','ST_Test3_L_max_overlap',
['ST_Test3_vars_' current_case(1:end-7)]
),'SSNR_in_all','SSNR_out_all','SNR_in_all','SNR_out_all','nr_factor_all','sd
_factor_all','z_v_filt_all','z_ST_all','FS','src','noise','Gain','current_cas
e','L','overlap');
    clear SSNR_in_all SSNR_out_all SNR_in_all SNR_out_all nr_factor_all
sd_factor_all z_v_filt_all z_ST z_v_filt SSNR_in SSNR_out H_ST SNR_in SNR_out
nr_factor sd_factor j


    % Run Wiener Filter Sweep
    z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
    z_W_all = zeros(max(length(noise),length(src)),length(L));
    for j = 1:length(L)
        L_current = L(j)
        [z_W, ~, ~, H_W, SSNR_in] = Weiner_filter( src', noise', L(j),
overlap(j)  );
        [ z_v_filt, SNR_in, SNR_out, SSNR_out, nr_factor, sd_factor, P_out,
P_out_n, P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src', noise',
L(j),overlap(j), H_W, z_W );


        SSNR_in_all(j) = SSNR_in;
        SSNR_out_all(j) = SSNR_out;
        SNR_in_all(j) = SNR_in;
        SNR_out_all(j) = SNR_out;
        nr_factor_all(j) = nr_factor;
        sd_factor_all(j) = sd_factor;
        z_v_filt_all(1:length(z_v_filt),j) = z_v_filt;
        z_W_all(1:length(z_W),j) = z_W;
    end
```

```
    %save output metrics to appropriate Test 3 folder
    save(fullfile('Weiner Filter','Weiner_Test3', ['Weiner_Test3_vars_'
current_case(1:end-7)]
),'SSNR_in_all','SSNR_out_all','SNR_in_all','SNR_out_all','nr_factor_all','sd
_factor_all','z_v_filt_all','z_W_all','FS','src','noise','Gain','current_case
','L','overlap');
    clear SSNR_in_all SSNR_out_all SNR_in_all SNR_out_all nr_factor_all
sd_factor_all z_v_filt_all z_W z_v_filt SSNR_in SSNR_out H_W SNR_in SNR_out
nr_factor sd_factor j

    %write all filtered output signals to wav files in the appropriate folder
    for j = 1:length(L)
        wavwrite((1./(max(abs(z_ST_all(:,j))))).*z_ST_all(:,j),FS,
fullfile('BF out wavs', [current_case(1:end-7) '_STBF_L' num2str(L(j))]))
        wavwrite((1./(max(abs(z_W_all(:,j))))).*z_W_all(:,j),FS, fullfile('BF
out wavs', [current_case(1:end-7) '_WBF_L' num2str(L(j))]))
    end
    clear current_case
end
```

Weiner_filter.m

```
function [z_W, R_vv, R_yy, H_W, SSNR_in] = Weiner_filter( y, v_only, L,
overlap  )
%Applies a multi channel weiner filter noise reduction algorithm to a multi
%channel input given a noise only segment and a signal + noise segment
%   y = NxP matrix to be beamformed. Rows correspond to each mic channel and
columns
%       correspond to sample numbers. N = # of mics, P = length of data
%       NOTE: y is not time shifted or aligned. Raw data from mics
%   v_only = NxQ matrix of each channel of noise recording with no speech
%       present. Does not need to be same length as y_k. Used to calculate
%       Rvv
%   Fs = sampling rate of data
%   L = filter length to be used in algorithm
%   overlap = sample overlap for each length L frame


if(overlap>L)
    error('Sample block "L" must be larger than sample overlap')
end
N = size(y,1); % Calculate number of mics in array

% Calculate "U" matrix for future H_w calculations
U = zeros(L,N*L);
U(1:L,1:L) = eye(L,L); % U = [I(LxL) 0(LXL)... 0(LXL)] (LXNL matrix)


%%% Initialize Variables
R_vv = zeros(N*L,N*L); % Intialize noise correlation matrix NLxNL
m = 0; % intialize noise correlation matrix calculation counter for averaging
later
P_in_n_SEG =0;
R_yy = zeros(N*L,N*L);
```

```
n = 0;
P_in_SEG =0;


%%% Estimate statistics by breaking inputs into frames of samples
for i = 1:L-overlap:max(length(v_only),length(y)) %initialize index to start
at beginning of each block of length L taking sample overlap into account

    %%%% Calculate R_vv from v_only %%%%
    if(L+i-1 <= length(v_only))
    v = v_only(:,i:L+i-1)'; % Take a block of L samples at starting at
current index i

    %Organize NxL matrix containing sample blocks into NLx1 matrix

    v_L = v(:);


    %Calculate current correlation matrix for N blocks of L samples
    m = m+1; % number of times correlation matrix is calculated

    %Average correlation matrix over time
    R_vv = ((m-1)/m)*R_vv + (v_L * v_L')./m;
    %R_vv = lambda*R_vv + (v_L * v_L').*(1-lambda);
    P_in_n_SEG = ((m-1)/m)*P_in_n_SEG + mean(var(v))./m;
    end

    %%%% Calculate R_yy for each L sample block of (signal + noise) data %%%%
    if(L+i-1 <= length(y))
    y_L = y(:,i:L+i-1)'; % Take a block of L samples at starting at current
index i

    %Organize NxL matrix containing sample blocks into NLx1 matrix
    y_k = y_L(:);

    n = n+1;
    R_yy = ((n-1)/n)*R_yy + (y_k*y_k')./n;
    %R_yy = lambda*R_yy + (y_k*y_k').*(1-lambda);
    P_in_SEG = ((n-1)/n)*P_in_SEG + mean(var(y_L))./n;
    end

end

SSNR_in = (P_in_SEG-P_in_n_SEG)./P_in_n_SEG;


%%%% Calculate Filter matrix H_w and execute filtering operation %%%%
H_W = ((eye(N*L,N*L) - (R_yy\R_vv))*U')';


for i = 1:L-overlap:length(y)
    if(L+i-1 > length(y)) %if current block will exceed length of input array
data, break for loop
```

```
        break
    end
    y_L = y(:,i:L+i-1)'; % Take a block of L samples at starting at current
index i

    %Organize NxL matrix containing sample blocks into NLx1 matrix

    y_k = y_L(:);

    z_W(i:i+L-1,1) = H_W*y_k;

end

end
```

Weiner_Simulation_test1.m

```
%%% Weiner Simulation test 1 %%%
% varying noise cutoff frequencies with spacial correlation
% with block lengths also varied
prec = [ .956   0   .445;
         0      0   .458;
         -.915  0   .445;
         .956   0   1.368;
         0      0   1.358;
         -.915  0   1.278;
         .956   0   2.139;
         0      0   2.133;
         -.915  0   2.075];

psrc = [0 40 0];
[ src, ~, src_pwr ] = sim_3D( bftest0, FS, psrc, prec );

L = [2     3     4     5     6     7     8    10    16    20    24    28
30    32];
overlap = L - 1;
fc = [ 50 100 200 300 500 800 1000 2000 3000 ];
for h = 1:length(fc)
    current_fc = fc(h)
    [ noise_corr, R_vv ] = plane_noise_R_3D( prec, src_pwr, length(src),
fc(h),7,FS );
    y = src+noise_corr;

    for i = 1:length(L)
    L_current = L(i)
    [z_W, ~, ~, H_W] = Weiner_filter( y, noise_corr, L(i), overlap(i)  );
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( y, noise_corr, L(i),overlap(i),
H_W, z_W );

    SNR_in_all(h,i) = SNR_in;
    SNR_out_all(h,i) = SNR_out;
    nr_factor_all(h,i) = nr_factor;
```

```
    sd_factor_all(h,i) = sd_factor;
    P_out_all(h,i) = P_out;
    P_out_n_all(h,i) = P_out_n;
    P_sig1_all(h,i) = P_sig1;
    P_out_sig1_all(h,i) = P_out_sig1;

    end
end
```

Weiner_Test1_Filt_length.m

```
%% Weiner Filter Test 1- Varying filter lengths with 0 overlap

L = [2     3     4     5     6     7     8    10    16    20    24    28
30    32];
overlap = 0;
z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
z_W_all = zeros(max(length(noise),length(src)),length(L));
for i = 1:length(L)
    L_current = L(i)
    [z_W, ~, ~, H_W] = Weiner_filter( src, noise', L(i), overlap(i)  );
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src, noise', L(i),overlap(i), H_W,
z_W );

    SNR_in_all(i) = SNR_in;
    SNR_out_all(i) = SNR_out;
    nr_factor_all(i) = nr_factor;
    sd_factor_all(i) = sd_factor;
    P_out_all(i) = P_out;
    P_out_n_all(i) = P_out_n;
    P_sig1_all(i) = P_sig1;
    P_out_sig1_all(i) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_W_all(1:length(z_W),i) = z_W;
end
```

Weiner_Test2_overlap.m

```
%% Weiner Filter Test 2- Varying sample overlaps with L =  10 filter

L(1:10) = 10;
overlap = 0:9;
z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
z_W_all = zeros(max(length(noise),length(src)),length(L));
for i = 1:length(L)
    current_overlap = overlap(i)
    [z_W, ~, ~, H_W] = Weiner_filter( src', noise', L(i), overlap(i)  );
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src', noise', L(i),overlap(i), H_W,
z_W );

    SNR_in_all(i) = SNR_in;
    SNR_out_all(i) = SNR_out;
```

```
    nr_factor_all(i) = nr_factor;
    sd_factor_all(i) = sd_factor;
    P_out_all(i) = P_out;
    P_out_n_all(i) = P_out_n;
    P_sig1_all(i) = P_sig1;
    P_out_sig1_all(i) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_W_all(1:length(z_W),i) = z_W;
end
```

Weiner_Test3_filt_length_max_overlap.m

```
%% Weiner Filter Test 3- Varying filter lengths with max overlap
tic

L = [2     3     4     5     6     7     8    10   16  20 24 28 30 32 ];
overlap = L-1;
z_v_filt_all = zeros(max(length(noise),length(src)),length(L));
z_W_all = zeros(max(length(noise),length(src)),length(L));
for i = 1:length(L)
    L_current = L(i)
    [z_W, ~, ~, H_W, SSNR_in] = Weiner_filter( src', noise', L(i), overlap(i)
);
    [ z_v_filt, SNR_in, SNR_out, SSNR_out, nr_factor, sd_factor, P_out,
P_out_n, P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src', noise',
L(i),overlap(i), H_W, z_W );


    SSNR_in_all(i) = SSNR_in;
    SSNR_out_all(i) = SSNR_out;
    SNR_in_all(i) = SNR_in;
    SNR_out_all(i) = SNR_out;
    nr_factor_all(i) = nr_factor;
    sd_factor_all(i) = sd_factor;
    P_out_all(i) = P_out;
    P_out_n_all(i) = P_out_n;
    P_sig1_all(i) = P_sig1;
    P_out_sig1_all(i) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_W_all(1:length(z_W),i) = z_W;
end
toc
```

Weiner_Test4_N.m

```
%% Weiner Filter Test 4- Varying Mic #s

N = 1:9;
L = [2 3 4 5 6 7 8 10 16 20 24 28 32] ;
overlap = L-1;
z_v_filt_all = zeros(max(length(noise),length(src)),length(N));
z_W_all = zeros(max(length(noise),length(src)),length(N));
for i = 1:length(N)
```

```
    current_receiver_num = N(i)

    for j= 1:length(L)

    [z_W, ~, ~, H_W] = Weiner_filter( src(:,1:N(i))', noise(:,1:N(i))', L(j),
overlap(j)  );
    [ z_v_filt, SNR_in, SNR_out, nr_factor, sd_factor, P_out, P_out_n,
P_sig1, P_out_sig1 ] = BB_Filter_Metrics( src(:,1:N(i))', noise(:,1:N(i))',
L(j),overlap(j), H_W, z_W );

    SNR_in_all(i,j) = SNR_in;
    SNR_out_all(i,j) = SNR_out;
    nr_factor_all(i,j) = nr_factor;
    sd_factor_all(i,j) = sd_factor;
    P_out_all(i,j) = P_out;
    P_out_n_all(i,j) = P_out_n;
    P_sig1_all(i,j) = P_sig1;
    P_out_sig1_all(i,j) = P_out_sig1;
    z_v_filt_all(1:length(z_v_filt),i) = z_v_filt;
    z_W_all(1:length(z_W),i) = z_W;
    end
end
```

x_beam_plot.m

```
function [ beam_pwr ] = x_beam_plot( x, Fs, prec, x_scan, y_plane, z_plane,
W)
%[ beam_pwr ] = x_beam_plot( x, Fs, prec, x_scan, y_plane, z_plane, W)
%   x = raw (un-aligned) data from mic array (NxL)
%   Fs = sampling rate of source signal
%   prec = postion of receivers [x1 y1 z1;x2 y2 z2...]
%   x_scan = array of points used to scan the x axis
%   y_scan = defines the y plane for the x direction sweep
%   z_scan = defines the z plane for the x direction sweep
%   W = weighting vector for each mic channel

%   beam_pwr = vector of calculated signal power at each x_scan point
%   src_pwr = calculated source power over all inoput samples




 %% Find Max power point
 for i = 1:length(x_scan)                           %look direction
     [ ~, ~, pout_scan ] = DS_beamformer( x, Fs, [x_scan(i) y_plane z_plane],
W, prec );
     beam_pwr(i) = pout_scan;
 end

 %% Plot Beam Power vs X axis
 figure ()

 subplot(2,1,1)
```

```
 plot(x_scan, beam_pwr);
 xlabel('x position (m)')
 ylabel('beam power (W)')

 subplot(2,1,2)
 plot(x_scan, 10.*log10(beam_pwr))
 xlabel('x position (m)')
 ylabel('beam power (dB)')


end
```