SASP HOMEWORK REPORT

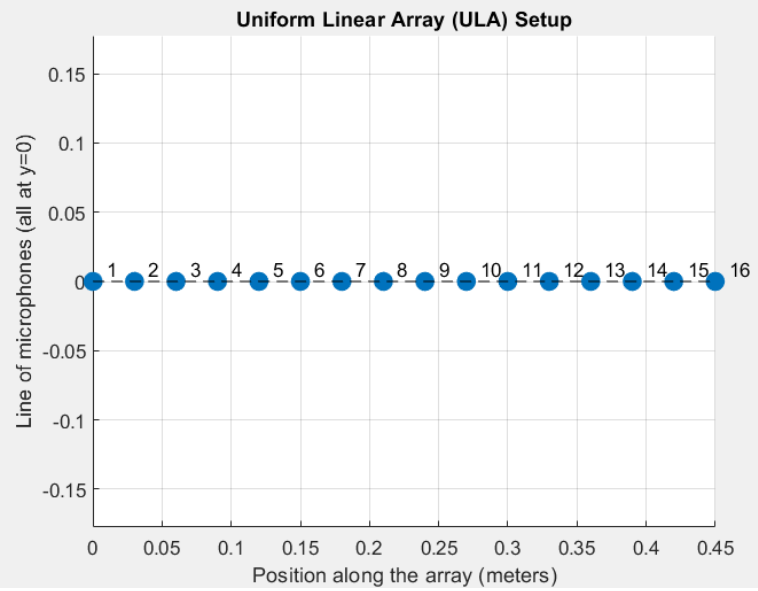
Angelo Antona, Alessandro Manattini

SOUND ANALYSIS, SYNTHESIS, AND PROCESSING Module 1

1 CONTENTS

2 PROJECT OVERVIEW

This project aims to implement Direction of Arrival (DOA) estimation using audio signal processing techniques. The source localization involves a multichannel recording acquired using a uniform linear microphone array (ULA) composed of 16 MEMS microphones spaced along a 45 cm length. This array captures audio data via an Audio-over-IP connection at a sampling rate of 8 kHz. The acoustic scene features a moving sound source in front of the ULA, and the speed of sound is established at 343 m/s.

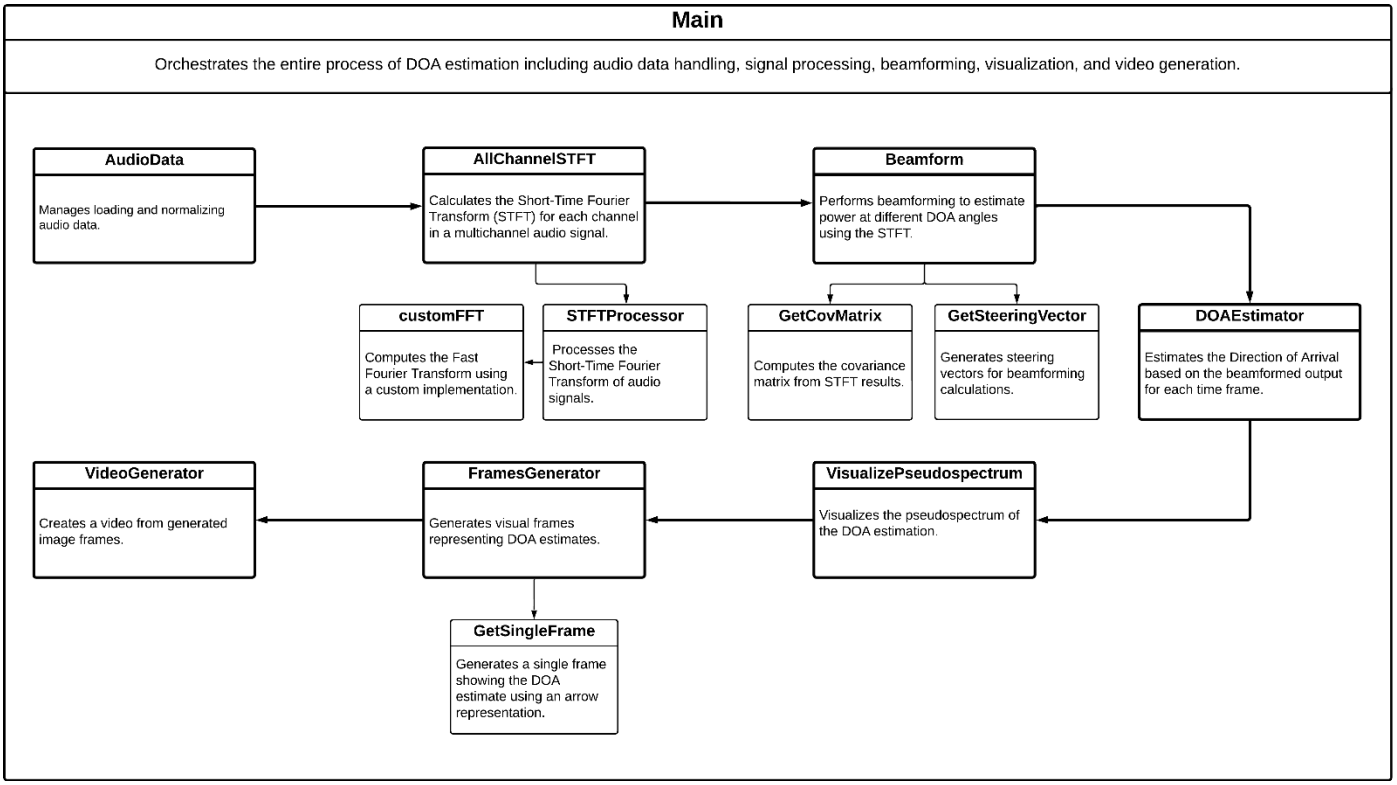


Our objective is to implement a delay-and-sum beamformer tailored for localizing wide-band sources. Given the narrow-band nature of spatial filtering, the project handles each frequency band independently and performs spatial filtering frame by frame to accommodate the source's time-varying nature.

3 DEVELOPMENT AND TESTING APPROACH

The development strategy follows a bottom-up approach, starting with fundamental components and progressively integrating and testing higher-level functions. Each component is validated through dedicated test cases (included in the *MainTest.m* script) that verify its performance individually before proceeding to system-level integration.

The flow of usage for the classes/functions is illustrated in the figure below.



In the next section of the report, each of these classes will be analyzed individually.

4 COMPONENT DESCRIPTIONS AND TESTS

4.1 AUDIODATA CLASS

4.1.1 Functionality

Handles audio data loading, storage, and normalization. It ensures data is appropriately scaled for processing.

Properties	Methods
• **'Data'**: stores the audio samples loaded from a file. • **'SampleRate'** : stores the sample rate of the audio data.	• **function obj = AudioData(filepath)**→ It is the constructor of the class. It uses the function '*audioread'* to load audio data from the specified '*filepath'*. The loaded audio samples and their corresponding sample rate are stored in the '*Data'* and '*SampleRate'* properties, respectively. • **function obj = normalize(obj)**→ normalizes the audio data stored in the '*Data'* property so that the maximum absolute amplitude is exactly 1.

4.1.2 AudioData Test in MainTest

Checks file existence, loads audio, and normalizes it. The test verifies correct data loading and effective normalization by comparing pre and post-normalization amplitude levels.

Command window output:

```
Audio data info before normalization:
Sample Rate: 8000
Number of Samples: 117864
Number of Channels: 16
9 samples from the first channel:
  1.0e-03 *
   0.0916  -0.0916   0.7935  -0.3967  -0.2441  -0.4272   0.3662   0.0916  -0.5493   0.6714
--------------------------------------------------------------------------------------------------------------------------------------------------------
Audio data info after normalization:
Max value: 1
Same 9 samples from the first channel after normalization:
   0.0146  -0.0146   0.1268  -0.0634  -0.0390  -0.0683   0.0585   0.0146  -0.0878   0.1073
>>
```

4.2 CUSTOMFFT FUNCTION

4.2.1 Functionality

The 'customFFT' function implements a recursive Fast Fourier Transform (FFT) algorithm tailored to input lengths that are powers of two (Cooley-Tukey algorithm). It showcases classic divide-and-conquer strategy used in FFT computations.

Input	Output
• **x:** a vector representing the time-domain signal samples. The length of 'x' must be a power of two.	• **X**: a vector representing the frequency-domain spectrum of the input signal 'x'.

4.2.2 Implementation Details

Recursive Decomposition: The function splits the input signal 'x' into two parts:

- X_even: contains the samples from the even indices of 'x'.
- X_odd: contains the samples from the odd indices.

This step reduces the problem size by half, reducing the complexity. The recursion is applied to both 'X_even' and 'X_odd'.

```
% Recursive FFT computation
X_even = customFFT(x(1:2:end));
X_odd  = customFFT(x(2:2:end));
```

Combination using the "Butterfly": After the recursive calls return the FFTs of the even and odd indexed parts, the results are combined using the "butterfly" operations typical of the FFT algorithms. This involves complex exponential multipliers, which are precomputed for efficiency. The combination is performed using the code lines in the figure X.

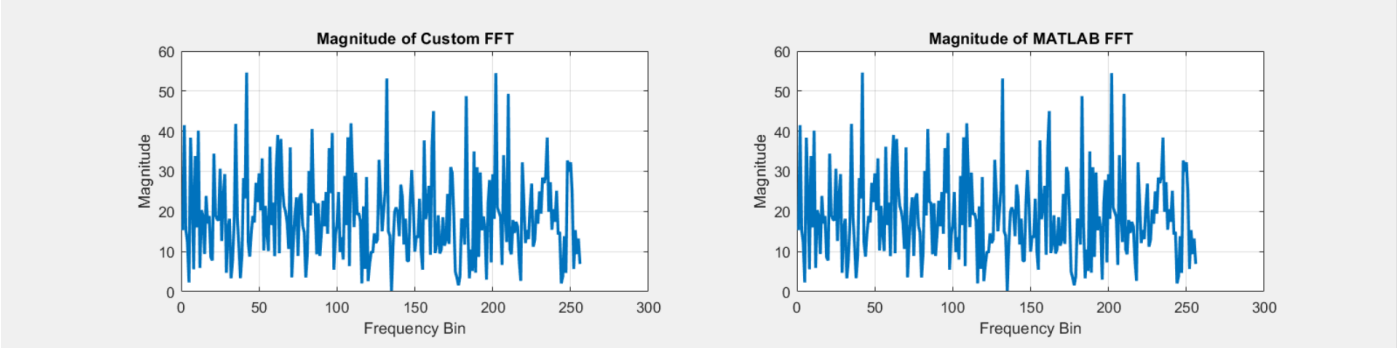
```
factor = exp(-2i * pi * (0:N/2-1) / N);
X = [X_even + factor .* X_odd, X_even - factor .* X_odd];
```

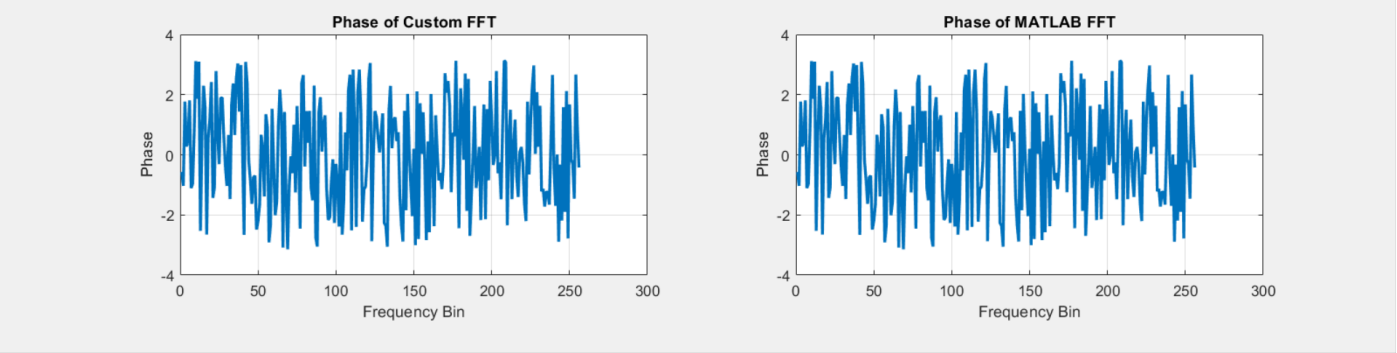
Recursion termination:

- **Base Case:** The recursion terminates when the length of 'x' is 1, since the FFT of it is 'x' itself. This is the simplest form of the DFT, where the transform of a single sample is the sample.
- **Error Handling:** The function checks if the length of 'x' is a power of two. If not, it raises an error. This requirement is crucial in the **Cooley-Tukey algorithm**.

4.2.3 CustomFFT Test in MainTest

Compares the output of **CustomFFT** against MATLAB's built-in **fft** function, ensuring the custom implementation's accuracy through magnitude and phase comparisons.





Command window output:

```
Norm of the difference between custom and MATLAB FFT: 1.336132e-13

The custom FFT implementation matches MATLAB's fft function within the tolerance.

>>
```

4.3 STFTProcessor Function

4.3.1 Functionality

The **'STFTProcessor'** function calculates the STFT for a single channel signal. It includes the windowing, overlapping and FFT processes.

Inputs	Outputs
• **x:** input signal. • **fs:** sampling frequency of the audio signals. • **window:** window function applied to each frame of the STFT. • **overlap:** a scalar indicating the proportion of overlap between consecutive frames. • **nfft**: the number of FFT points to compute the STFT.	• **S:** STFT matrix. • **f:** frequency axis. • **t:** time axis.

4.3.2 Implementation Details

Windowing and Overlap: Computes the number of overlapping and applies the window to each frame of the signal.

FFT Computation: It uses the 'customFFT' function that handles padding and computes the FFT for each frame.

Spectrum Handling: It stores only the positive half of the frequency spectrum in the STFT matrix.

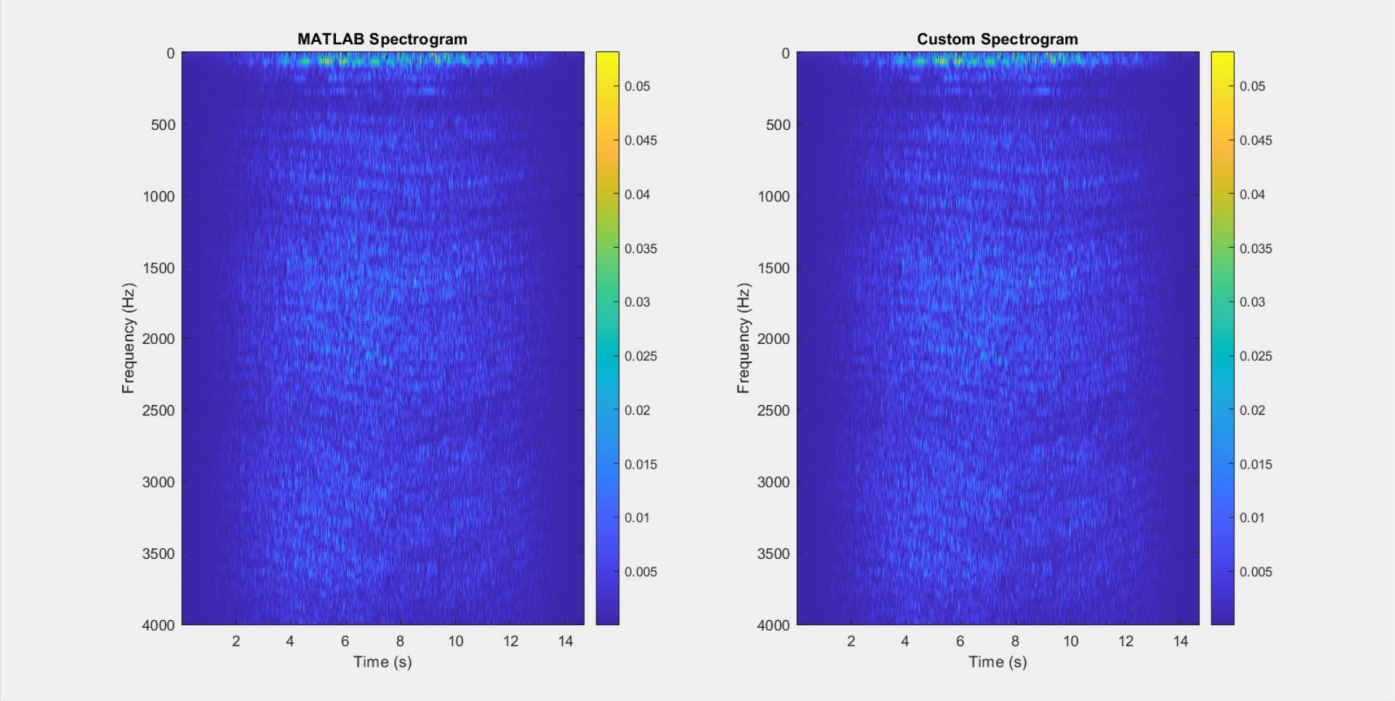
```
% Apply the STFT for each frame
for i = 1:num_frames
    % sample indices of the current frame
    idx = (i - 1) * overlap_length + (1:window_length);

    % Apply the window to the current frame
    frame = x(idx) .* window;
    % Zero-pad the frame to the next power of two
    % (customFFT can manage only signals with length = n^2)
    padded_frame = [frame', zeros(1, nfft - window_length)];

    % Calculate the frame FFT with zero-padding
    X = customFFT(padded_frame);
    % Store only the positive half of the frequency spectrum
    S(:, i) = X(1:num_freqs);
end
```

4.3.3 STFTProcessor Test in MainTest

Verifies the custom STFT's correctness against MATLAB's **spectrogram** function by comparing the output matrices for discrepancies.



4.4 AllChannelSTFT Function

4.4.1 Functionality

Extends **STFTProcessor** to handle multichannel audio data, computing STFTs for each channel independently.

Inputs	Outputs
• **signal:** a matrix where each column represents an audio signal from one microphone. • **fs:** sampling frequency of the audio signals. • **window:** window function applied to each frame of the STFT. • **overlap:** a scalar indicating the proportion of overlap between consecutive frames. • **nfft**: the number of FFT points to compute the STFT. • **MicrophoneCount**: number of channels in the multichannel signal.	• **S_multi**: a 3D array containing the STFT results for each channel. • **f:** the frequency axis for the STFT. • **t:** the time axis for the STFT.

4.4.2 Implementation Details

As showed in the code in figure X, the function performs the following operations:

Channel Verification: The function first checks if the number of columns in the 'signal' matrix matches 'MicrophoneCount' to confirm correct signal dimensions.

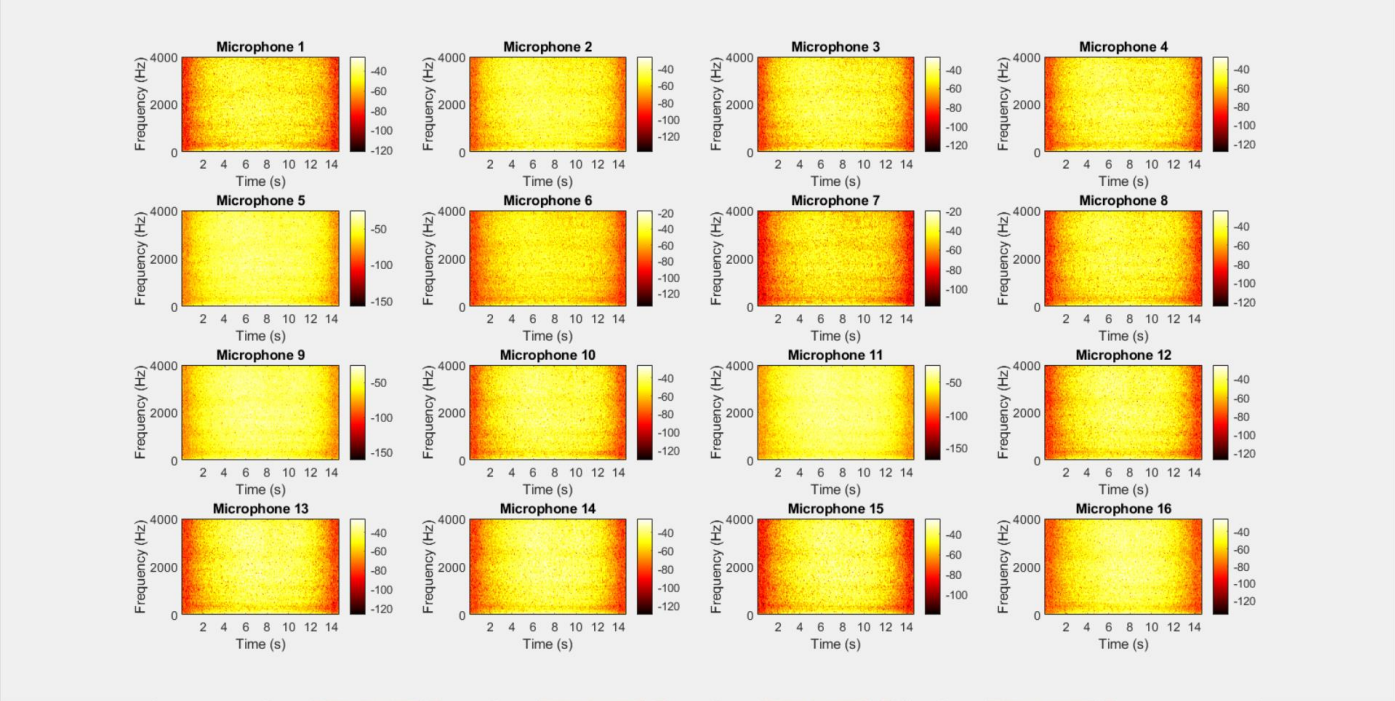
STFT Calculation: For each microphone channel, the function calls 'STFTProcessor' to compute the STFT.

Output Initialization: It initializes a 3D array 'S_multi' after computing the first channel's STFT to store all subsequent results.

```
% Process each channel individually
for mic = 1:MicrophoneCount
    [S, f, t] = STFTProcessor(signal(:, mic), fs, window, overlap, nfft);
    % If this is the first microphone, initialize the 3D STFT array
    if isempty(S_multi)
        S_multi = zeros(size(S, 1), size(S, 2), MicrophoneCount);
    end
    % Store the result in the corresponding "slice" of the 3D array
    S_multi(:, :, mic) = S;
end
```

4.4.3 AllChannelSTFT Test in MainTest

Ensures that the function accurately processes each microphone channel without errors and integrates the results into a 3D array.



4.5 GetCovMatrix Function

4.5.1 Functionality

The ‘GetCovMatrix’ function computes the spatial covariance matrix of signals captured by an array of microphones at different frequencies.

Input	Output
• **S_time**: a 2D matrix representing the STFT results for all microphones and frequencies at a specific time frame.	• **R**: a 3D array where each slice along the third dimension represents the covariance matrix for a particular frequency.

4.5.2 Implementation Details

As showed in the picture X, the function works in the following way:

Frequency Iteration: It processes each frequency band individually.

Matrix Construction: For each frequency, it calculates the outer product of the signal vector with itself to form the covariance matrix.

```
for f = 1:numFreqs
        % Extract the signals at frequency f for all microphones
        S_f = S_time(f, :).'; % Column vector [numMics x 1]
        % Outer product and accumulation for single time frame
        R(:, :, f) = S_f * S_f';
end
```

4.5.3 GetCovMatrix Test in MainTest

Ensures that the matrices are Hermitian, affirming the correctness of the covariance computations.

```
La matrice di covarianza alla frequenza 1 è hermitiana.
La matrice di covarianza alla frequenza 2 è hermitiana.
La matrice di covarianza alla frequenza 3 è hermitiana.
La matrice di covarianza alla frequenza 4 è hermitiana.
La matrice di covarianza alla frequenza 5 è hermitiana.
La matrice di covarianza alla frequenza 6 è hermitiana.
La matrice di covarianza alla frequenza 7 è hermitiana.
La matrice di covarianza alla frequenza 8 è hermitiana.
La matrice di covarianza alla frequenza 9 è hermitiana.
La matrice di covarianza alla frequenza 10 è hermitiana.
>>
```

4.6 GetSteeringVector Function

4.6.1 Functionality

The function 'GetSteeringVector' generates the steering vector for a microphone array, which is used in beamforming to focus the array's sensitivity in a specific direction relative to the sound source.

Inputs	Output
• **theta:** DOA angle in degrees. • **d:** spacing between microphones in the array. • **c:** speed of sound. • **numMics**: number of microphones in the array. • **freq**: frequency at which the steering vector is calculated.	• **a:** steering vector as a column vector.

4.6.2 Implementation Details

Wavenumber Calculation: It computes the wavenumber based on the frequency and the speed of sound.

```
k = 2 * pi * freq / c;
```

Vector Computation: It uses the DOA angle and the microphone spacing to calculate the phase shift for each microphone, resulting in the steering vector.

```
a = exp(-1i * k * d * sin(deg2rad(theta)) * (0:(numMics-1))).';
```

4.7 BEAMFORM FUNCTION

4.7.1 Functionality

Implements beamforming to enhance the signal from a specific direction using the steering vectors. It aggregates the energy from different directions to pinpoint the sound source's location.

Inputs	Output
• **S**: a 3D array containing the STFT results obtained in the function 'AllChannelSTFT' (called S_multi). • **d**: the spacing between microphones in the array, measured in meters. • **c**: the speed of sound. • **Fs**: the sampling frequency. • **numMics**: number of microphones in the array. • **theta_range**: array of angles (in degrees) for which the DOA estimates are computed → from -90° to 90°.	• **p_theta_time**: a 2D matrix where each element represents the computed power for a specific angle and time frame, indicating the likelihood of sound arriving from that direction.

4.7.2 Implementation Details

Parameter Initialization: It extracts the number of frequencies and time frames from the input data and it also initializes the frequency vector.

```
[numFreqs, numTimeFrames, ~] = size(S);
p_theta_time = zeros(length(theta_range), numTimeFrames);
freqs = linspace(0, Fs/2, numFreqs); % Frequency vector outside the loop
```

Time Frame Processing: For each time frame, it extracts the corresponding STFT data for all frequencies and microphones.

```
% Extract the slice for the current timeFrame across all freq and micS
S_time = squeeze(S(:, timeIdx, :));% S_time dimensions=[numFreqs x numMics]
```

Covariance Matrix Calculation: It computes the spatial covariance matrix for the signals received by the microphone array. The covariance matrix is computed calling the function 'GetCovMatrix'.

```
R = GetCovMatrix(S_time);
```

Angle Processing: For each angle in 'theta_range', the function computes the beamforming power using a steering vector, which is computed calling the function 'GetSteeringVector'.

```
for i = 1:length(theta_range)
  for f = 1:numFreqs
    % Compute the steering vector for the current angle and frequency
     a = GetSteeringVector(theta_range(i), d, c, numMics, freqs(f));
     % Beamforming power accumulation
     p_theta_time(i, timeIdx) = p_theta_time(i, timeIdx)+abs(a'
*R(:,:,f)*a);
  end
  % Normalize the accumulated power by the number of microphones squared
  p_theta_time(i, timeIdx) = p_theta_time(i, timeIdx) / numMics^2;
end
```

Normalization: At the end the function normalizes the power values across all angles and time frames to facilitate comparison.

```
% Normalize the output power over all time frames
max_p = max(p_theta_time(:));
if max_p > 0
    p_theta_time = p_theta_time / max_p;
end
```

4.7.3 Test in MainTest

Covered implicitly through **DOAEstimator** tests, validating beamforming effectiveness in estimating directions.

4.8 DOAESTIMATOR FUNCTION

4.8.1 Functionality

The 'DOAEstimator' function is designed to determine the DOA of sound based on the beamforming output computed in the 'Beamform' function. This function identifies the angle at which the received power is maximized, suggesting the most likely direction from which the sound originated.

Inputs	Output
• **p_theta_time**: a matrix (number of angles x number of time frames) containing the power values for each angle and time frame. This matrix is the output from the beamforming process, where each row corresponds to a different DOA angle, and each column corresponds to a different time frame. • **theta_range**: an array of DOA angles (from -90° to 90°) → it corresponds to the rows in 'p_theta_time'. These angles represent the possible directions from which the sound may arrive.	• **doa_estimates**: an array containing the estimated DOA for each time frame. Each element in this array represents the angle from which sound is inferred to have arrived during the corresponding time frame.

4.8.2 Implementation Details

Parameter Initialization: It initializes an array 'doa_estimates' to store the DOA estimates for each time frame. The length of the array must be equal to the number of time frames.

```
% Initialize the output array to store DOA estimates
numTimeFrames = size(p_theta_time, 2);
doa_estimates = zeros(1, numTimeFrames);
```

Time Frame Analysis:

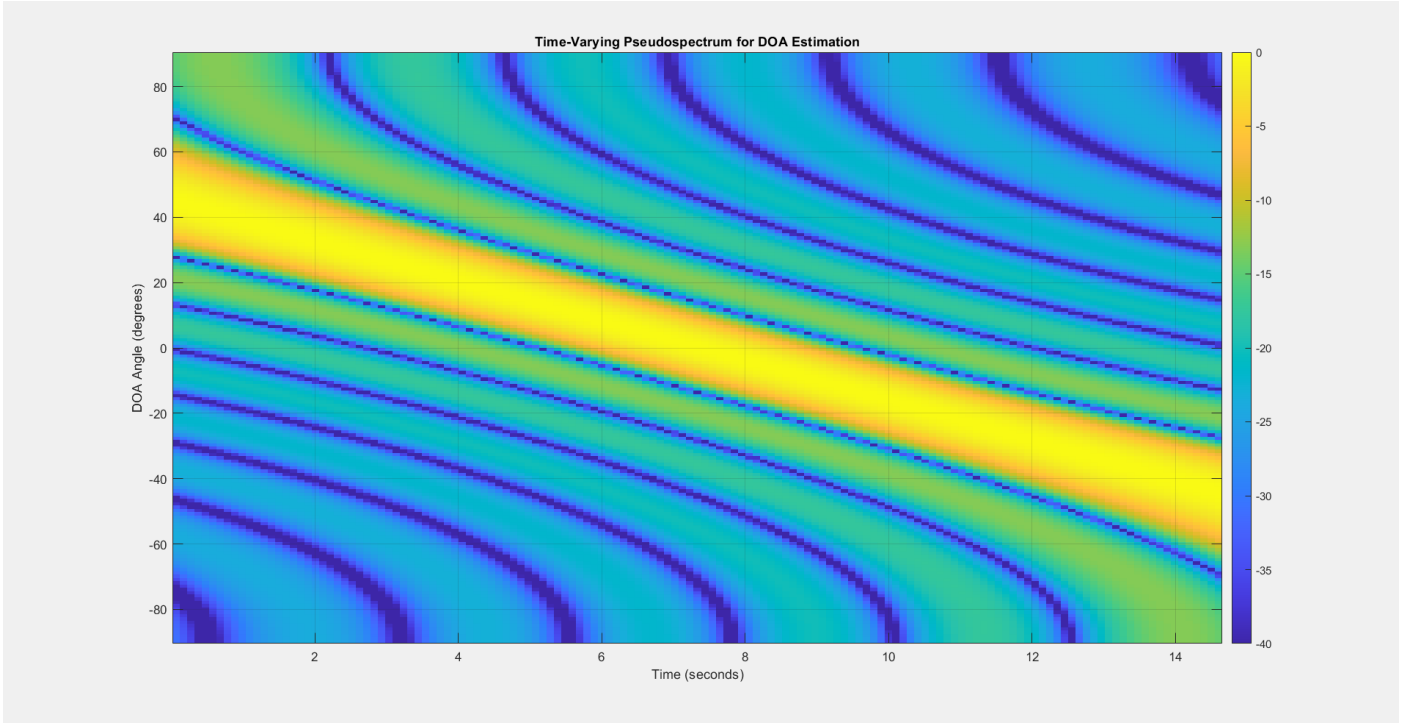
- **Loop through time frames**: the function iterates over each time frame to analyze the power distribution across different angles.
- **Maximum power detection**: for each time frame, the function identifies the index of the maximum power in the 'p_theta_time' matrix. This index corresponds to the row (angle) where the highest power is detected.

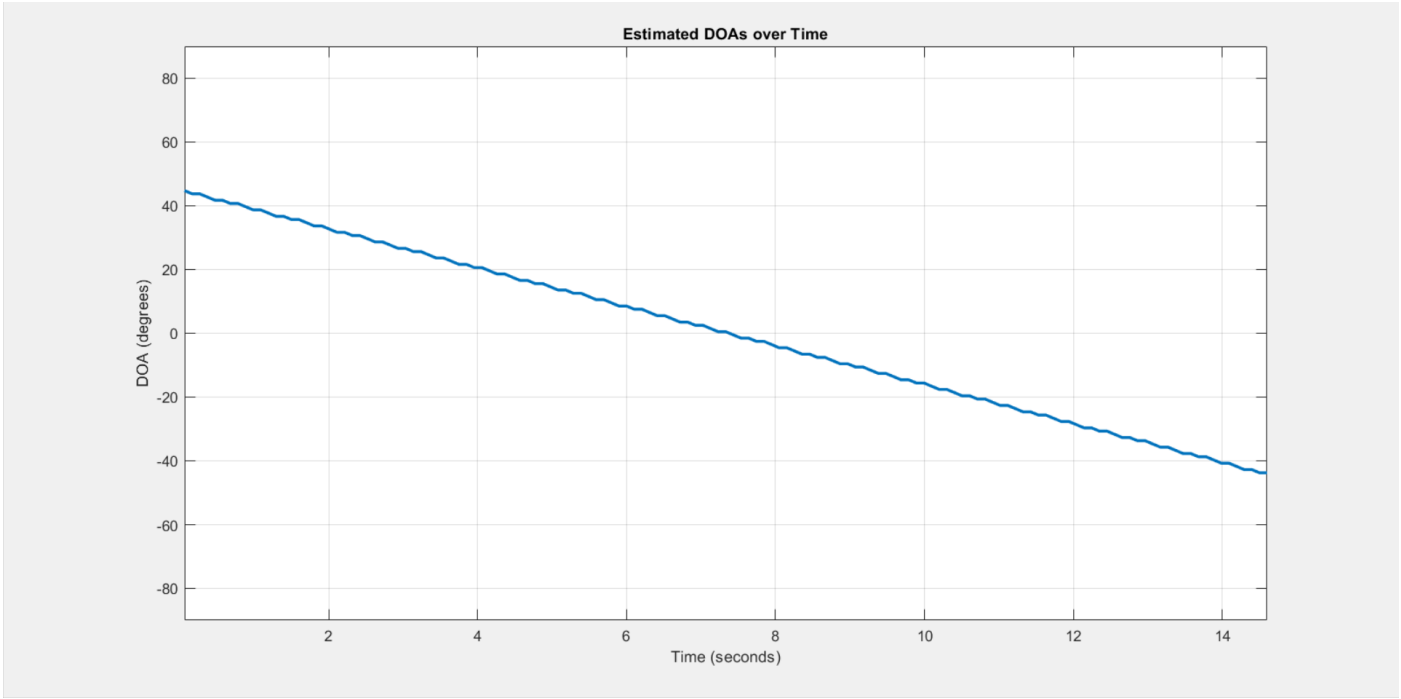
- **Angle association**: the angle corresponding to this index is retrieved from ‘theta_range’ and recorded as the DOA estimate for that time frame.

```
% Loop through each time frame to find the angle with the maximum power
for timeIdx = 1:numTimeFrames
    % Find the index of the maximum power at this time frame
    [~, maxIdx] = max(p_theta_time(:, timeIdx));
    % Store the corresponding angle as the DOA estimate
    doa_estimates(timeIdx) = theta_range(maxIdx);
end
```

4.8.3 Test in MainTest

Integrates tests for **Beamform**, **GetSteeringVector**, and **GetCovMatrix** by using them to calculate the DOA estimates over time of a sweeping sound source that moves from 45 degrees to -45 degrees relative to the array's front, verifying the end-to-end functionality of the DOA estimation process.





4.9 Visualization Functions (visualizePseudospectrum and visualizeDOAestimates)

These functions graphically represent the results of the DOA estimation process, providing a visual interpretation of how DOA varies over time and the pseudospectrum of detected angles.

4.10 Video Generation (getSingleFrame, FramesGenerator, VideoGenerator)

These functions generate visual frames and compile them into a video, illustrating the DOA estimation process over time. They create a dynamic visualization of the estimated angles as they evolve, enhancing understanding and presentation of the results.

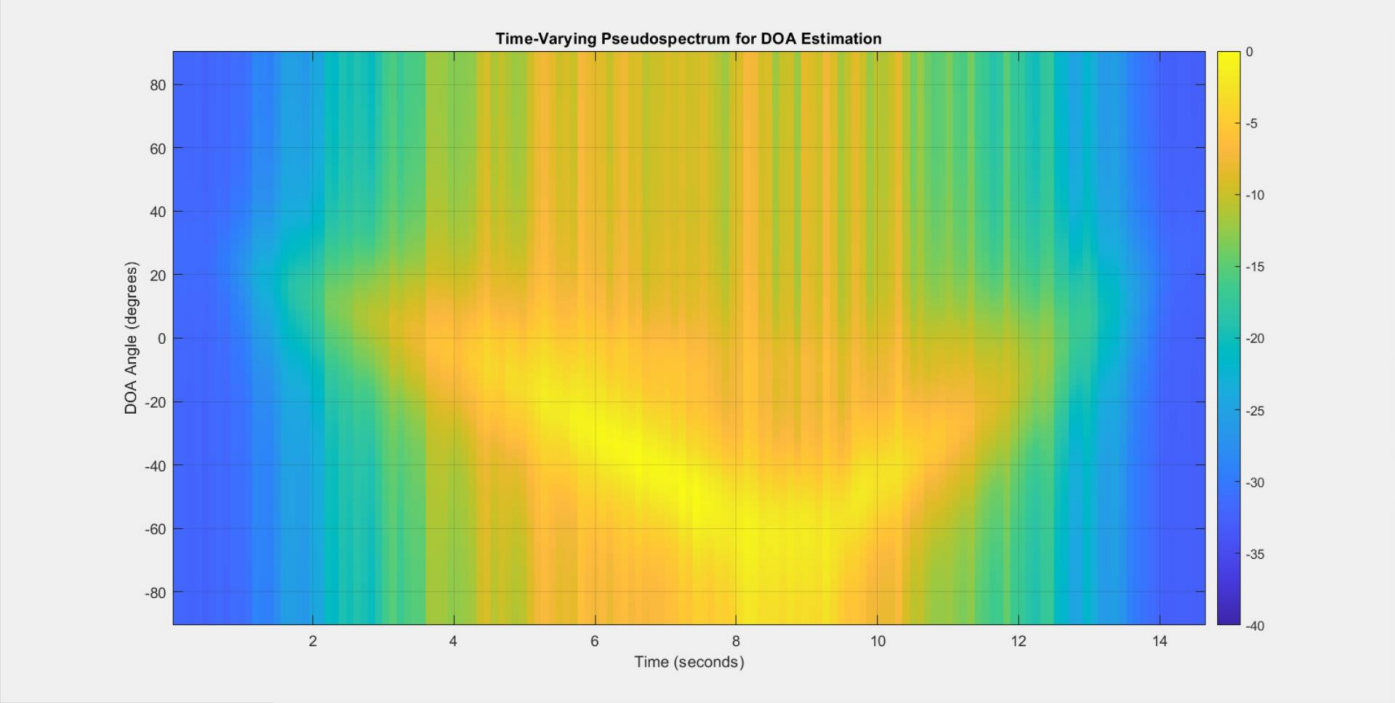
4.11 Main Script Functionality

The main script orchestrates all the components to implement the DOA estimation process. It loads audio data, performs STFT, applies beamforming and DOA estimation, and visualizes the results through plots and video.

5 Results Of The Provided Signal Analysis

5.1 Analysis of the Pseudospectrum

The "Time-Varying Pseudospectrum for DOA Estimation" graph provides a comprehensive view of the power distribution across various directions (DOA angles) over time, allowing us to visualize how the direction of arrival of sound energy varies as the source moves.



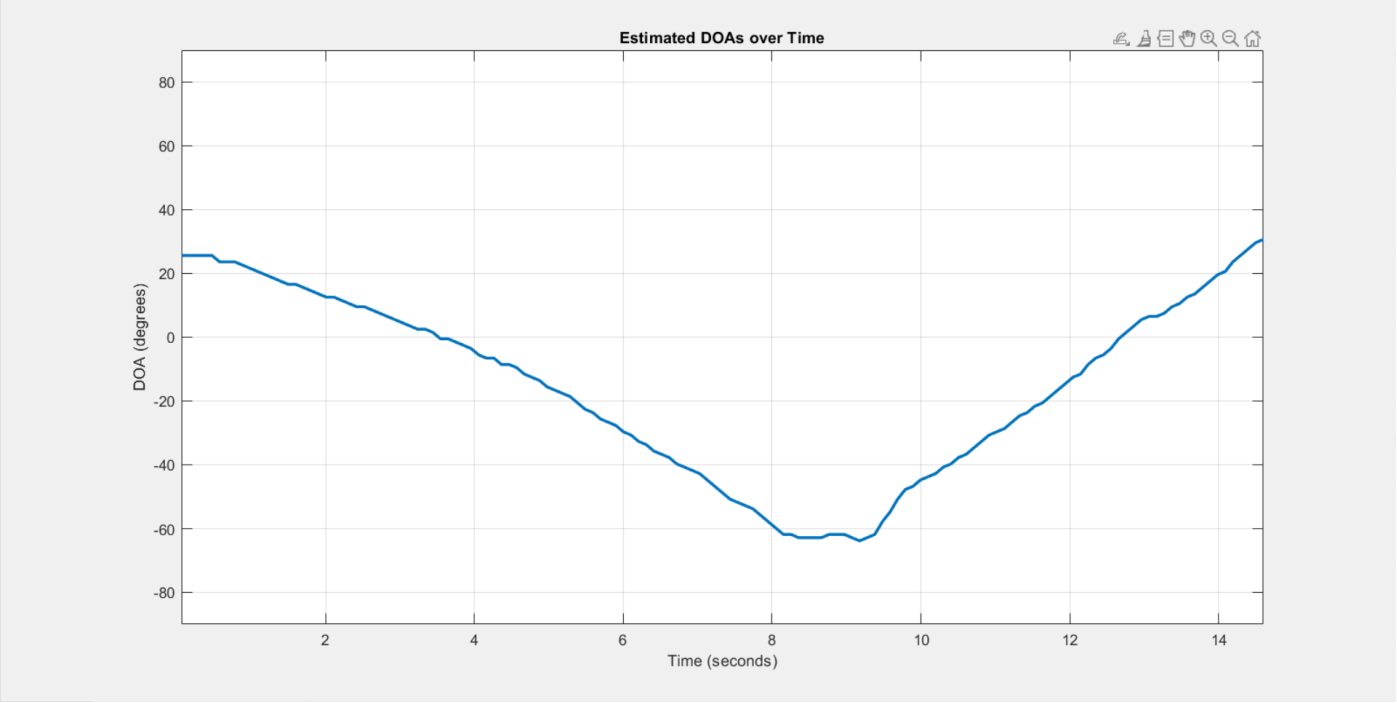
The brightest areas, indicating higher power levels, are concentrated around the center of the angle range (close to 0°). This suggests that the most likely direction of the sound source is directly in front of the microphone array.

There is a noticeable shift in intensity across different times.

At certain times, for example at 8 seconds, there is a broader spread of moderate intensities across a wider range of angles. This could indicate moments where the sound source is moving or there is interference from multiple directions.

5.2 Analysis of DOA Estimates

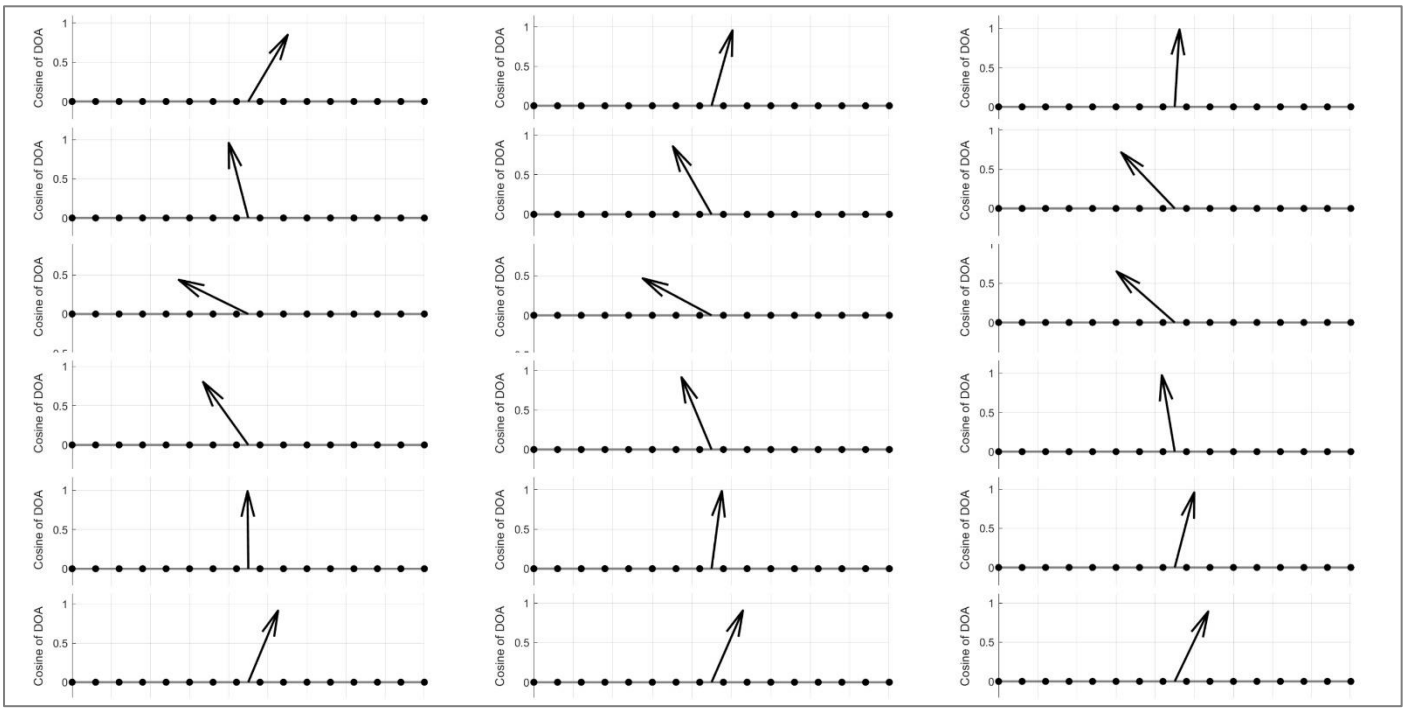
The graph below presents the estimated Direction of Arrival (DOA) of the sound source.



Since the sound source is moving, the DOA estimates plot displays that the DOA starts at a higher angle, drops to a significant low and then rises again. The graph begins at around 20° and experiences a steady decline to below -60°. The lowest point around the 100th sample suggests a moment where the sound source is at its farthest position relative to the initial orientation of the microphone array.

After the lowest point, there is a sharp recovery in the DOA estimates, where the angle swiftly increases, possibly indicating that the sound source is moving back towards its original starting direction.

We can also visualize the DOA estimates through the following image, which depicts a sequence of some frames from the video generated by the VideoGenerator function. This frame-by-frame visualization allows us to observe the temporal evolution of the direction of arrival estimates, highlighting how the system dynamically tracks the position of the sound source over time.



5.3 CONCLUSION

In conclusion, the analysis results affirm the effectiveness of the designed acoustic source localization system.