Acoustic Warfare

Marcus Allen, Tuva Björnberg, Isac Bruce,
Ivar Nilsson, Mika Söderström, Jonas Teglund

July 2023

1 Introduction

This is a project that performs real-time beamforming applied to sound waves to visualize audio sources in a room on top of a webcam feed and also listen to audio in a given direction. The audio data from the microphone arrays is sampled by the FPGA on a Zybo-Z7 20 development board and passed through via Ethernet to a PC. The PC then processes the sampled data and scans the room with the beamforming algorithm to locate the audio sources. These sources are used to generate a heatmap that is layered on top of a camera feed for visualization. It is also possible to click on a certain location on the camera feed which causes the beamforming algorithm enhance the audio coming from that angle. This project is a continuance on last years summer project Acustic Warfare where the FPGA-sampling was done with one microphone array and the beamforming algorithm was not implemented in real-time. These parts can be read about in the reports, Acoustic-Warfare: FPGA sampling by Ivar Nilsson and Jacob Drotz [ND], Phased microphone planar arrays by Lucas Åkerstedt and Mika Söderström [ÅS].

2 Background

The Zybo Z7-20 is a development board with a dual-core ARM Cortex-A9 processor and FPGA (Field Programmable Gate Array). With multiple I/O-ports such as Jtag and Ethernet a PC can uphold communication with the FPGA, ARM processors and with the case of this project, microphone arrays by way of Pmod ports.

The 8x8 microphone array have 64 ICS-52000 microphones with a signal converter of analog to digital, an anti-aliasing filter, a wide-band response and omni-directional directionality. The microphones are daisy chained in four groups of 16 and the sampling starts with an incoming signal, the WS-pulse. With a frequency of 48828,125 Hz the microphone outputs 24 bits of data on the rising edge of a 25 MHz external clock called sck, into 32-bit TDM-slots (Time-Division Multiplexing slots). The remaining 8 bits is padded according to two's compliment and concatenated. This is the first part of the PL (programmable logic) of the FPGA.

The PS (processing system) on the other hand is run by the ARM core. It collects data from the PL and enables a PC to receive said data by an Ethernet UDP-socket.

On the PC side, Ubuntu is used as operating system and here the microphone samples are being processed to generate a heatmap using different beamforming algorithms.

3 Beamforming

3.1 Delay and sum beamforming

One common beamforming technique for spatial filtering is delay-and-sum beamforming, also known as conventional beamforming. The method is based on delaying the individual signals from the elements and then adding them together. We should then be able to spatially filter incoming signals such that there is an increase in power and sensitivity in the desired direction, and suppression of signals arriving from all other directions. The increase in signal power is due to a constructive interference between the delayed signals coming from a desired direction, and the decrease because of destructive interference between signals from other directions.

The introduced delay for each microphone signal will be determined by the difference in time that it will take for the sound to arrive to the different microphone elements. For this beamforming algorithm, the sound is assumed to be a plane wave, which is true if we are in the far-field of the operating microphone array.

The incoming signal from microphone $i = 0, 1, ..., M-1$, where M is the number of microphones, is denoted $x_i(n)$, where $n = 1, 2, ..., N$ and N the number of samples. Then, the signal is delayed using some delay operator $\mathcal{D}_i$ for the specific microphone i according to

$$y_i(n) = \mathcal{D}_i\{x_i(n)\}. \tag{1}$$

The delay of the signal could either be performed in the frequency domain by changing the phase of the signal, or in the time domain by simply delaying the signal a specific time. Then, all delayed signals for each microphone are added together to one signal

$$z(n) = \frac{1}{M}\sum_{i=0}^{M-1} y_i(n). \tag{2}$$

This signal $z(n)$ is the spatially filtered signal in the desired direction. The power of the filtered signal is then found using

$$P = \frac{1}{N}\sum_{n=1}^{N} |z(n)|^2. \tag{3}$$

The signal $z(n)$ can be used to listen to the spatially filtered sound from a desired direction. If we want to visualize the power from the sound sources from different directions, the Eq. (1)-(3) are repeated for all directions.

3.1.1 Phase delay, frequency domain

When delaying a signal in the frequency domain, the phase of the signal is changed. If the signals are Fourier transformed $\mathcal{F}$, a time delay of t_0 is equal to

$$\mathcal{F}\{x(t - t_0)\} = X(f)\mathrm{e}^{-\mathrm{j}\omega t_0} \tag{4}$$

where the phase ϕ_0 is calculated with

$$\phi_0 = \omega t_0 = -k\,\hat{\boldsymbol{r}}_\mathrm{s} \cdot \boldsymbol{r}_i' \tag{5}$$

Here, θ och φ are the angles for the "scanning" direction in the spherical coordinate system, $\hat{\boldsymbol{r}}_\mathrm{s}$ is the unity vector for the scanning direction, and $\boldsymbol{r}_i'$ is the position for microphone i. The phase delay ϕ_0 is frequency dependent, which means that signals with different frequencies can not be delayed the same time t_0 using the same phase.

3.1.2 Time delay, time domain

In the time domain, we do not consider the frequency of the incoming signal, but solely delay the signal a specific time. The exact time delay for a microphone is determined by the direction of the source $\hat{\boldsymbol{r}}_\mathrm{s}$, the position of the microphone $\boldsymbol{r}'_i$ and the speed of sound c according to

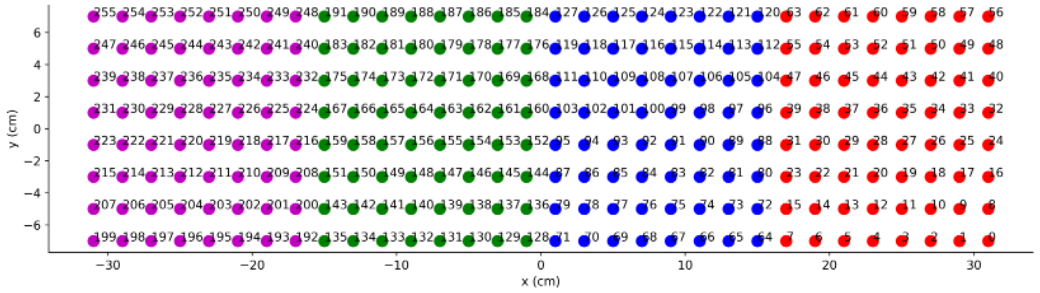
$$t_{\mathrm{d},i} = \frac{1}{c}\hat{\boldsymbol{r}}_\mathrm{s} \cdot \boldsymbol{r}'_i \tag{6}$$

$$s_{\mathrm{d},i} = f_\mathrm{s} t_{\mathrm{d},i} \tag{7}$$

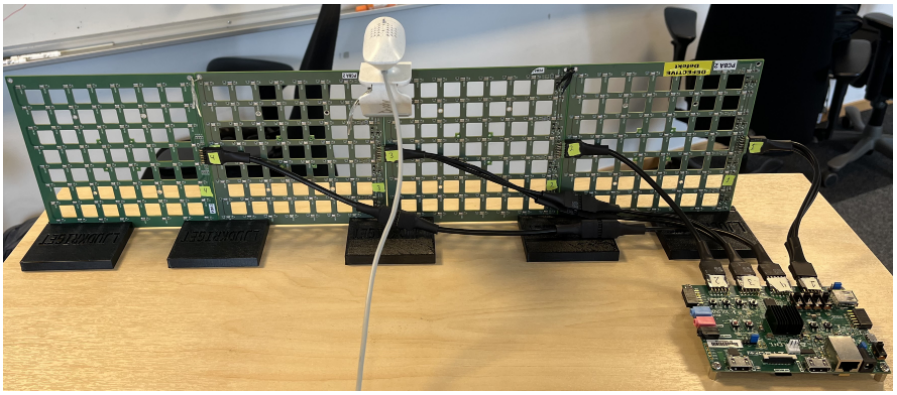
where $t_{\mathrm{d},i}$ is the delay in terms of seconds and $s_{\mathrm{d},i}$ the delay in terms of samples for microphone i. Since we have discrete signals, the delay might not be as exact as in the frequency domain.

3.2 Array setup

The array setup includes several individual arrays. One array is a planar 8x8 uniform phased array, with the microphone elements separated with a distance of $d = 2\,\mathrm{cm}$ in both directions. In the mathematical representation, the configuration of arrays are located in the xy-plane, at $z = 0$ with the center of the array configuration placed in the origin. The arrays have been stacked horizontally. This is because we want to achieve a higher resolution when spatially filter the signal in the horizontal direction to separate two humans talking, who most likely are on the approximately same vertical level. An illustration of the array setup with four arrays is presented in Fig. 1.



(a)



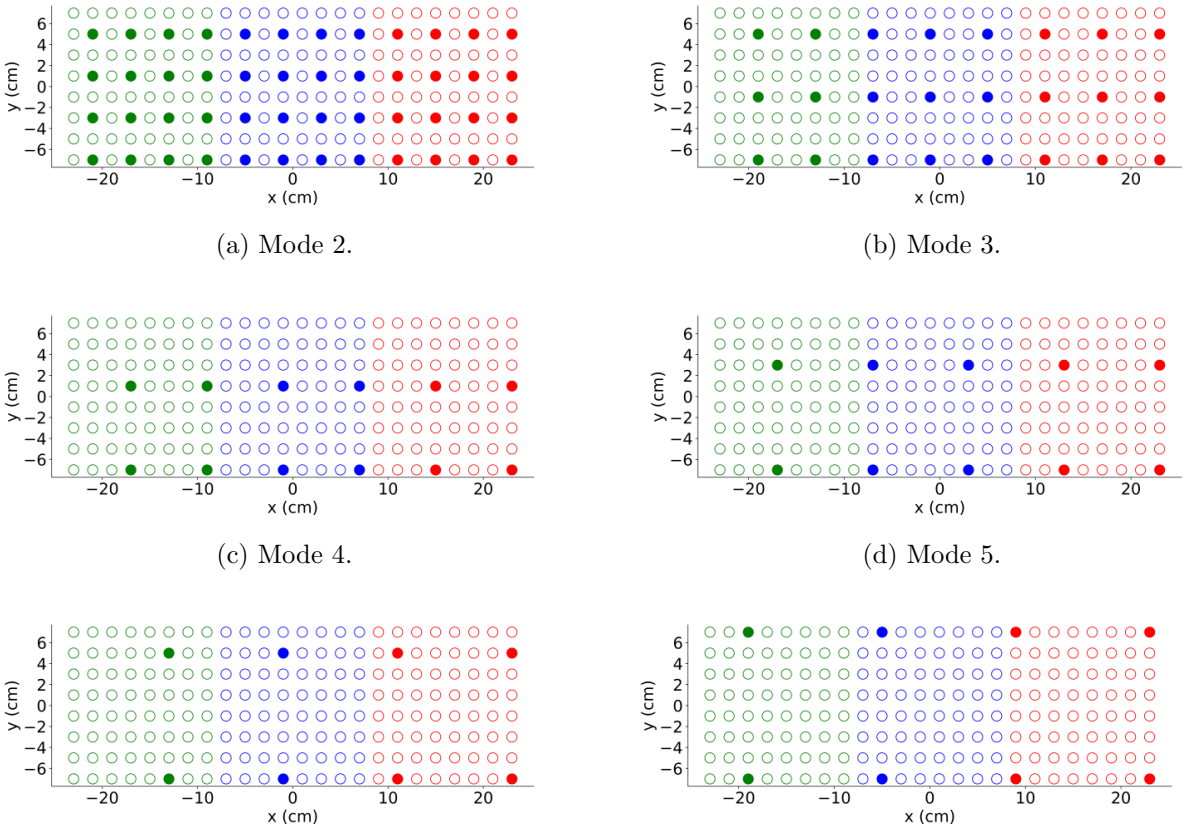
(b)

Figure 1: The array configuration, viewed from the backside of the arrays. (a) A representation of the array setyp, where, the dots represent the microphones, together with their indexes. The color indicates the individual arrays. Here, the arrays are placed next to each other without any separation. (b) The real array setup together with the camera.

For beamforming with arrays, the optimal distance between elements is equal to half the wavelength of the wave and therefore proportional to the frequency. This means that the distance of 2 cm between the elements are suitable for frequencies around 8 kHz. However, the spectrum of audible

sound is wide-band. Thus, we might benefit from not using all microphones and have introduced what we will call modes that only uses a selection of elements. The original setup seen in Fig. 1, using all microphones, is called mode 1. The next mode 2 uses every other microphone such that the spacing between the microphones is now $d = 4\,\text{cm}$, corresponding to half the wavelength of frequencies around 4 kHz. This is followed by mode 3 that uses every third microphone, etc. The different modes are presented in Fig. 2.

We introduce what we call an adaptive configuration, which means that the selected microphones, modes, used for the beamforming will be different depending on the frequency of the signal. How the individual modes and adaptive configuration will affect the antenna gain can be found under 3.3.



(a) Mode 2.

(b) Mode 3.

(c) Mode 4.

(d) Mode 5.

(e) Mode 6.

(f) Mode 7.

Figure 2: The adaptive configuration. All filled dot indicates that the microphones are used, all unfilled microphones are ignored in the specific modes in (a)-(f).

3.3 Array factor and antenna gain

The performance of an antenna is often characterized by the antenna gain, G, also referred to as antenna pattern. For isotropic elements (such as the microphones) in an array, the antenna pattern can be calculated as

$$G(\theta, \varphi) = |\text{AF}(\theta, \varphi)|^2 \tag{8}$$

where AF is the array factor, and θ and φ are angles in the spherical coordinate system [**en artikel**]. The array factor is found as

$$\text{AF}(\theta, \varphi) = \sum_{i=1}^{N} p_i e^{\mathrm{j}k\hat{\boldsymbol{r}}\cdot\boldsymbol{r}_i'} \tag{9}$$

where p_i is the phase shift for microphone i. When we have a main beam in the fixed direction $\hat{\boldsymbol{r}}_0$, the phase shift is given by

$$p_i = \mathrm{e}^{-\mathrm{j}k\hat{\boldsymbol{r}}_0 \cdot \boldsymbol{r}_i'} \tag{10}$$

The terms in the exponents of Eq. (9) and (10) can be written as

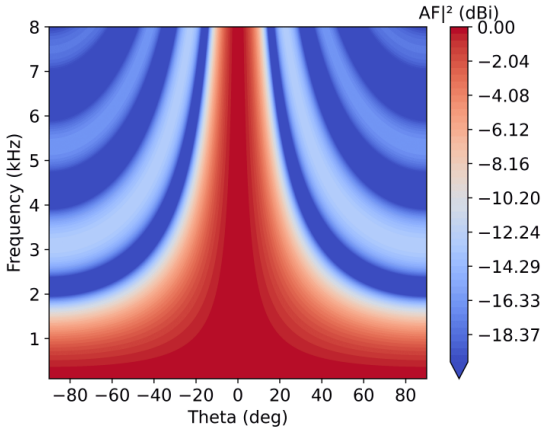
$$\hat{\boldsymbol{r}}_0 \cdot \boldsymbol{r}_i' = x_i \sin\theta_0 \cos\varphi_0 + y_i \sin\theta_0 \sin\varphi_0 \tag{11}$$

$$\hat{\boldsymbol{r}} \cdot \boldsymbol{r}_i' = x_i \sin\theta \cos\varphi + y_i \sin\theta \sin\varphi \tag{12}$$

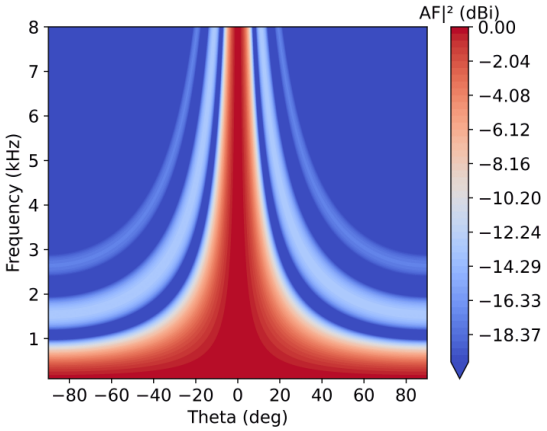
where the main beam is directed in the angles θ_0 and φ_0 and (x_i, y_i) are the coordinates of microphone i. The gain pattern obtained from the array will vary for different angles (θ, φ), and will be dependent on frequency.

3.3.1 Several arrays

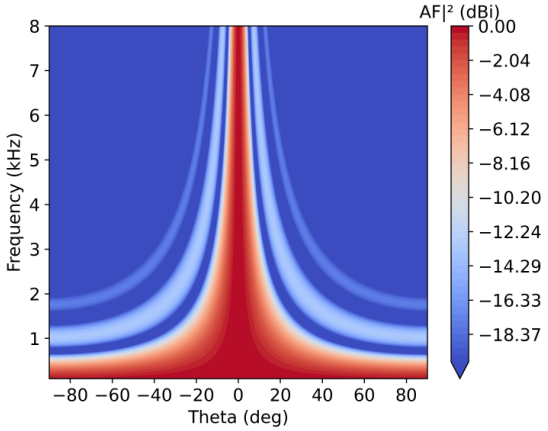
The gain pattern, Eq. (8), will change with different array setups. The configuration of arrays are based on section 3.2, where the arrays are placed along the horizontal axis. In the horizontal plane, where $\varphi = 0$ and $\theta \in [-90, 90]$, we obtain a normalized gain pattern as presented in Fig. 4. Here, the main beam is pointed in the direction of $\theta_0 = 0$, $\varphi_0 = 0$. For all array setups, independent of the number of arrays, the normalized gain pattern in the vertical plane (where $\varphi = 90$ and $\theta \in [-90, 90]$) will always be as in Fig. 4(a). It can be seen that the antenna pattern changes with frequency. For low frequencies, the beam becomes very wide, resulting in a beamforming with low resolution. From the figures it is evident that the main beam becomes more narrow for higher frequencies and the use of several arrays.



(a) One array.

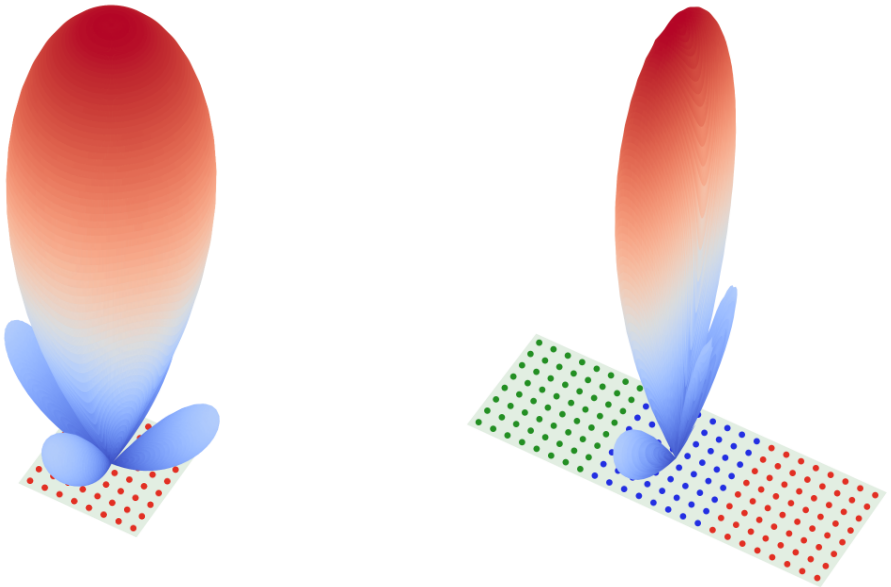


(b) Two arrays.



(c) Three arrays

Figure 3: The array factor for several arrays, using all microphones in each array. The arrays are placed along the horizontal axis, and the beam are pointed in the direction of $\theta_0 = 0$, $\varphi_0 = 0$.



(a) One array.

(b) Three arrays.

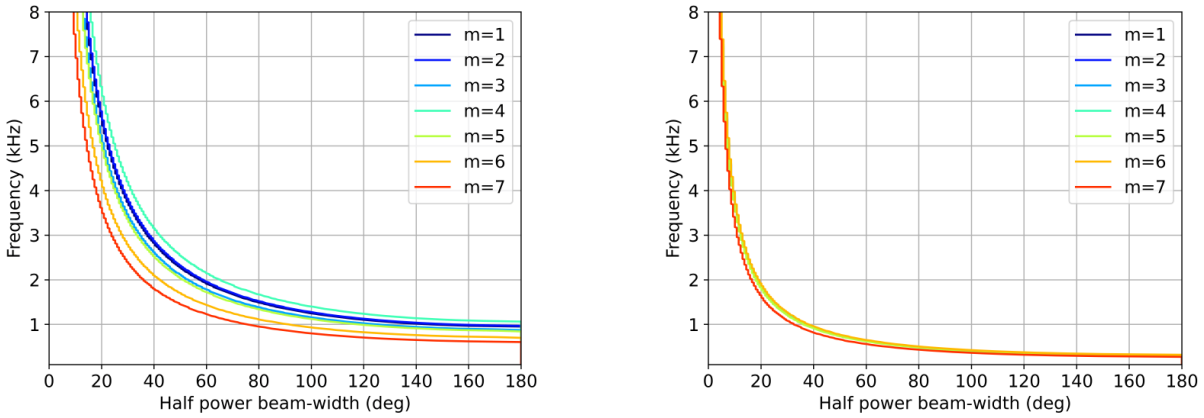
Figure 4: Antenna pattern in 3D, at a frequency of 4 kHz.

3.3.2 Modes and adaptive configuration

The antenna pattern will also vary with the different modes that have been described in section 3.2. For each mode, the half-power beam-width (HPBW) of the main beam have been calculated and is presented in Fig. 5.

For one array, as in Fig. 2(a), there is a larger difference between the HPBW of the different modes compared to three arrays, in Fig. 2 (b), where there is little to no difference. For the benefit of a more narrow main beam at lower frequencies, the adaptive configuration can be used when the array setup consist of one array. The adaptive configuration selects different modes in different parts of the frequency spectrum such that the distance between the active microphones never exceeds half the wavelength. The gain and HPBW for this case are presented in Fig. 6.

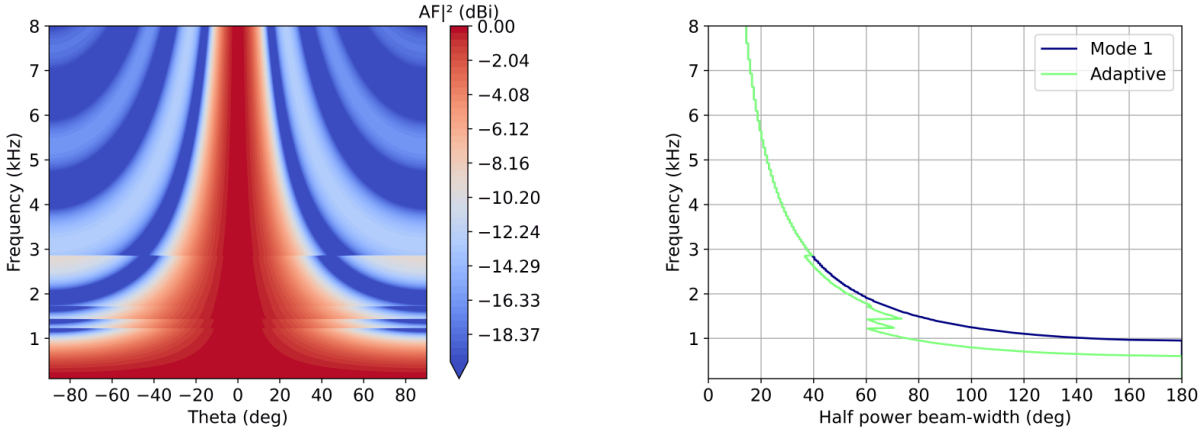
Even though the width of the main beam might not be reduced when having three arrays and the adaptive configuration, it could be beneficial to use the adaptive configuration. If the beamforming algorithm is using a method where the signal can be treated different depending on the frequency, the adaptive configuration could reduce the computation time since there is fewer element signals to process at the lower frequencies.



(a) One array.

(b) Three arrays.

Figure 5: The half-power beam-width of the main beam for the different modes.



(a)

(b)

Figure 6: Using adaptive configuration if only one array.

3.4 Scanning window and camera

Usually, spherical coordinates are used to describe the relation between the direction of the source and the position of a microphone $\hat{\boldsymbol{r}}_\mathrm{s} \cdot \boldsymbol{r}_i'$. In this specific project, it is desired to match the scanning angles of the beam to the camera image. Therefore, we use the Cartesian coordinates and find the term $\hat{\boldsymbol{r}}_\mathrm{s} \cdot \boldsymbol{r}_i'$ according to

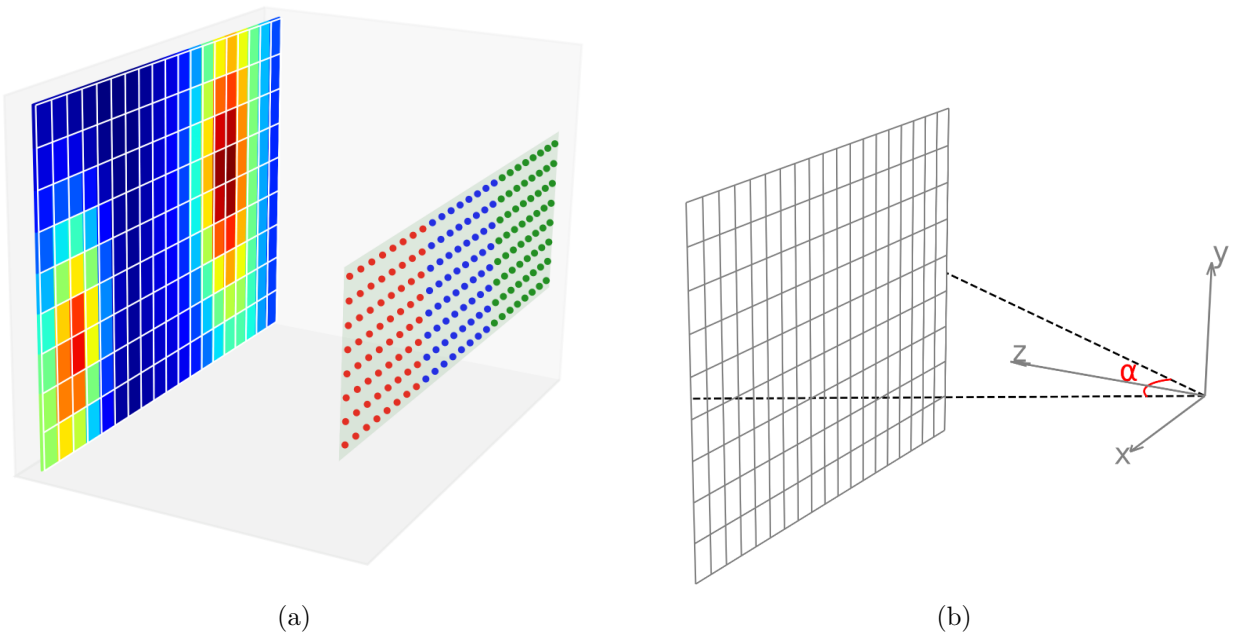
$$\begin{aligned}\hat{\boldsymbol{r}}_\mathrm{s} \cdot \boldsymbol{r}_i' &= \frac{\boldsymbol{r}_\mathrm{s}}{|\boldsymbol{r}_\mathrm{s}|} \cdot (x_i\hat{\boldsymbol{x}} + y_i\hat{\boldsymbol{y}}) \\ &= \frac{x_\mathrm{s}\hat{\boldsymbol{x}} + y_\mathrm{s}\hat{\boldsymbol{y}} + z_\mathrm{s}\hat{\boldsymbol{z}}}{\sqrt{x_\mathrm{s}^2 + y_\mathrm{s}^2 + z_\mathrm{s}^2}} \cdot (x_i\hat{\boldsymbol{x}} + y_i\hat{\boldsymbol{y}}) \\ &= \frac{(x_\mathrm{s}x_i + y_\mathrm{s}y_i)}{\sqrt{x_\mathrm{s}^2 + y_\mathrm{s}^2 + z_\mathrm{s}^2}} \end{aligned} \tag{13}$$

where $(x_\mathrm{s}, y_\mathrm{s}, z_\mathrm{s})$ are the coordinates of one point in the scanning window (i.e one scanning direction), and (x_i, y_i) are the coordinates of microphone i. The scanning window is defined with respect to the camera where the coordinates are determined by the aspect ratio of the camera, AS, and the horizontal angle of view α. Then, we find the relation between x_s, y_s and z_s according to

$$\mathrm{AS} = \frac{x_\mathrm{s}}{y_\mathrm{s}} \tag{14}$$

$$x_\mathrm{s} = z_\mathrm{s} \tan\left(\frac{\alpha}{2}\right) \tag{15}$$

In the relations that define the scanning window, Eq. (14) and (15), we have five variables (AS, *alpha*, x_s,y_s and z_s) where we set α and AS according to the camera. Then, we can choose z_s to something completely arbitrary and will have everything we need. For the camera used in this project we found the camera view angle to be approximately $\alpha = 59°$, and an aspect ratio of AS=16/9.



(a) (b)

Figure 7: An illustration of the defined scanning window together with the array configuration.

3.5 Implementation of the frequency domain beamforming

The algorithm for the frequency domain beamforming was implemented in Python, where the most computationally heavy parts of the algorithm were vectorized using the NumPy library.

3.5.1 Handling in-signals and performing the FFT

The beamforming algorithm expects to receive a signal, $x(n, \boldsymbol{r}_i')$, in the form of a two-dimensional NumPy array that has the shape NxM, shown in Eq. 16, where N is the number of samples in the signal and M is the number of microphones used in the array.

$$x(n, \boldsymbol{r}_i') = \begin{bmatrix} x(1, \boldsymbol{r}_0') & x(1, \boldsymbol{r}_1') & \cdots & x(1, \boldsymbol{r}_{M-1}') \\ x(2, \boldsymbol{r}_0') & x(2, \boldsymbol{r}_1') & \cdots & x(2, \boldsymbol{r}_{M-1}') \\ \vdots & \vdots & \ddots & \vdots \\ x(N, \boldsymbol{r}_0') & x(N, \boldsymbol{r}_1') & \cdots & x(N, \boldsymbol{r}_{M-1}') \end{bmatrix}. \tag{16}$$

The signals are also expected to contain strictly real data, which allows us to use the built-in discrete time "real" FFT function to reduce computational time, see X for more info. After the signal has been Fourier transformed it is multiplied with a phase-shift matrix in order to apply the required phase shift for beamforming.

3.5.2 Calculating the phase shift matrix

In order to determine in which direction to steer the beam, we need calculate the phase delay, ϕ_0, defined in Eq. (5). The phase delay is dependent on four variables: frequency, ν_j (discrete FFT), position of each microphone, $\boldsymbol{r}_i'$, the angle θ, and the angle φ, which means that the phase shift for all the combinations of these variables can be defined as a 4D-matrix. Using the definition of the scanning window in Eq. (13), we can also define the phase shift matrix in Cartesian coordinates, where θ and φ are replaced by x_s and y_s.

3.5.3 Outcome of the algorithm

When the signal $x(n, \boldsymbol{r}_i')$ has been Fourier transformed and phase shifted, its total power, P, is calculated over the scanning window by summarizing the power from all microphones, P_mics, over all frequencies, as the following:

$$X(\nu_j, \boldsymbol{r}_i') = \mathcal{F}_\mathrm{d}\{x(n, \boldsymbol{r}_i')\}, \tag{17}$$

$$Y(\nu_j, \boldsymbol{r}_i', x_\mathrm{s}, y_\mathrm{s}) = X(\nu_j, \boldsymbol{r}_i')\mathrm{e}^{-j\phi_0(\nu_j, \boldsymbol{r}_i', x_\mathrm{s}, y_\mathrm{s})}, \tag{18}$$

$$P_\mathrm{mics}(\nu_j, x_\mathrm{s}, y_\mathrm{s}) = \frac{1}{M}\left| \sum_{i=0}^{M-1} Y(\nu_j, \boldsymbol{r}_i', x_\mathrm{s}, y_\mathrm{s}) \right|^2, \tag{19}$$

$$P(x_\mathrm{s}, y_\mathrm{s}) = \sum_{j=1}^{N/2+1} P_\mathrm{mics}(\nu_j, x_\mathrm{s}, y_\mathrm{s}). \tag{20}$$

The resulting power can then be plotted as a heatmap over the scanning window. Note that the same be can be done to get $P(\theta, \varphi)$ by using the spherical definition of the phase shift matrix instead of the Cartesian. A block diagram showing the procedure of the beamforming algorithm can be seen in Fig. 8.

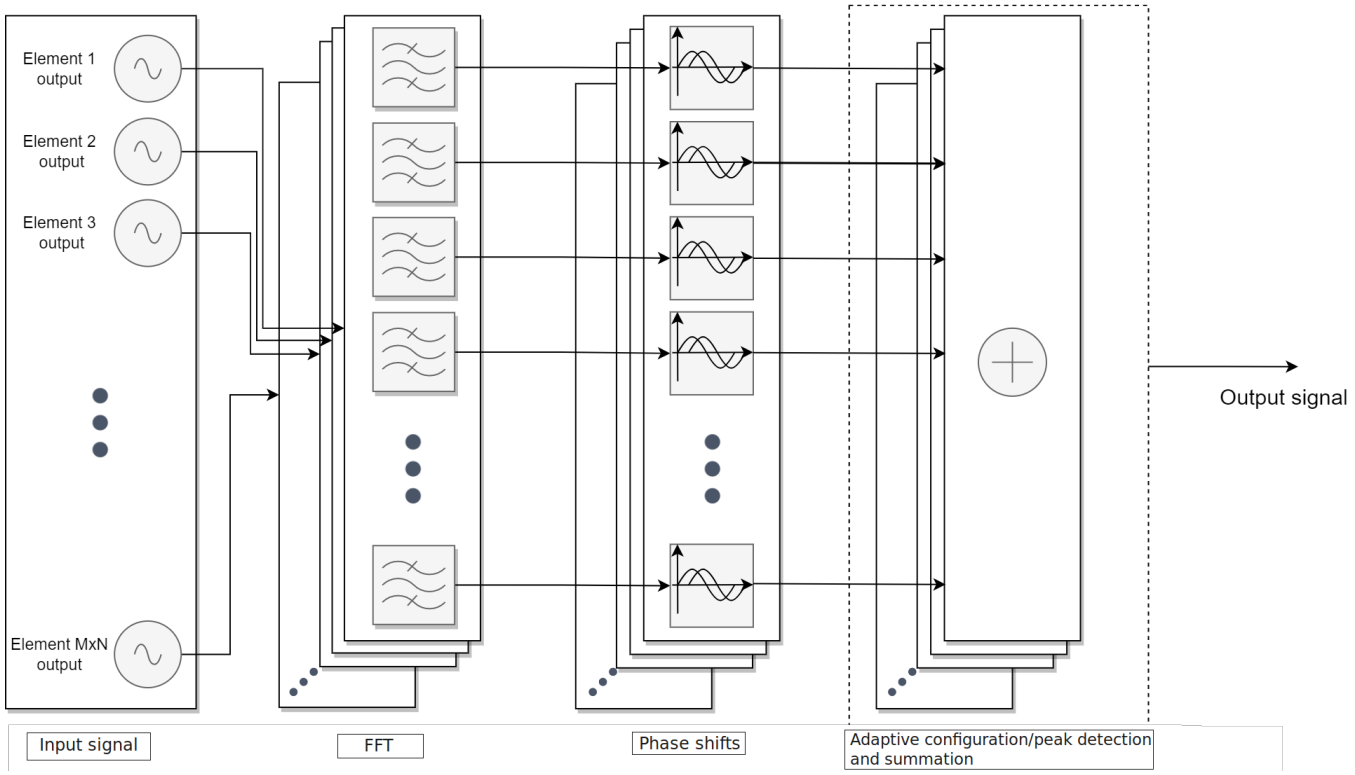


Figure 8: Block diagram of the beamforming algorithm receiving an input signal and outputting a heatmap visualizing the sources within the scanning window.

3.5.4 Optimization

When looking at the heatmap given by the beamforming algorithm we should expect to find a sound source at the peak of the beam. For frequencies that go further and further below 8 kHz, the beam will become wider and wider, which could cause problems when trying to detect exactly where a source located on the heatmap. This can be particularly problematic if there are several sources that emit sound at the same time, since the beams directed to each source will start to overlap if they are too wide. This could potentially create a new "false" peak in between the sources which obscures the real peaks since the false peak is higher than them, making it seem like there is only one source that isn't even located where any of the true sources are.

In order to try negate these problems, a peak detection function was implemented in the beamforming algorithm. Instead of summarizing $P_{\text{mics}}(\nu_j, x_s, y_s)$ over all frequency points, ν_j, the peak detection iterates through every frequency point. If it detects that the maximum value of P_{mics} at a frequency is above a certain threshold value, the position of that peak is marked out on a separate heatmap that is supposed to only show the exact locations of the peaks. However, in the case that there are multiple sources emitting sound at the same frequency point, the power from the sources may overlap and we face the same problem as before, at that frequency point.

Another way to find sources more distinctively and also reduce overlap between multiple sources is to use adaptive configuration as described in sections 3.2 and 3.3.2. It is also possible combine adaptive configuration with peak detection. This may potentially help the peak detection to distinguish multiple sources in situations where it is not able to on its own, by narrowing the beamwidth e.g. when two sources emit at the same frequency.

3.6 Implementation of time domain beamforming

The beamforing in the time domain is realised by delaying each microphone signal a time t_0. Here, the time delay are calculated in terms of samples according to Eq. (7), with the scanning window described in section 3.4. Since the signal arrives first to the microphone closest to the sound source this microphone will have the largest sample delay. For the mirophone furthest away from the source no delay will be introduced. The sample delays for all microphones and all scanning directions have been calculated in advance and stored in a three dimensional matrix $\mathrm{S}[x_\mathrm{s}, y_\mathrm{s}, i]$.

3.6.1 Pad

The signal can easily be delayed with whole samples by just padding the signal with zeros the correct number of samples. However, we will be limited in how precise we can delay the signal since we can not obtain a fractional samples. Hence, we can not delay the signal perfectly in any desired direction.

3.6.2 Convolution with sinc filter

A method based on interpolation, re-sampling and delaying of a signal using a fractional delay sinc filter can be used to delay a signal with non-integer samples. An input signal $x_i(n)$ can then be delayed with samples up to half the number of taps K of the filter. The filter coefficients $h_i(k)$ for a causal sinc shift are calculated for microphone i and scanning direction $\hat{\boldsymbol{r}}_\mathrm{s}$ according to

$$h_i(k) = \begin{cases} \mathrm{sinc}\left(k - \left[\frac{K-1}{2} + \tau_i\right]\right) , & |k| \leq K \\ 0 , & |k| > K \end{cases} \tag{21}$$

$$\mathrm{sinc}(w) = \frac{\sin(\pi w)}{\pi w} \tag{22}$$

with

$$\tau_i = 0.5 - s_{\mathrm{d},i} \tag{23}$$

where $s_{\mathrm{d},i}$ is the fractional sample delay for microphone i, calculated using Eq. (7) and (13). The filter coefficients h_i will we unique for each microphone and scanning direction.

Since we have a causal shift, the signal is shifted an additional number of $(N-1)/2$ samples. This means that the signal is actually delayed by the fractional delay $s_{\mathrm{d},i} + (N-1)/2$ after the convolution. To obtain a signal with correct delay $s_{\mathrm{d},i}$ the input signal will also be padded accordingly.

An ideal sinc function, Eq. (22), continues to infinity for both negative and positive w without dropping off to zero. Since we have a finite number of taps we will then have a discontinuity in the sinc function. Therefor, we multiply the filter coefficients with a blackman window to obtain an approximate ideal sinc filter. Additionally, the filter coefficients is normalised. Input signal will be filtered via convolution between filter coefficients h, to obtain the delayed signal

$$y_i(n) = \mathcal{D}_i\{x_i(n)\} = h_i * x_i \tag{24}$$

The convolution will then be performed for all microphones and added together according to Eq. (2). The power is calculated with Eq. (3).

3.6.3 Interpolation

Simple linear interpolation and re-sampling can be performed by the equation

$$y_i(n_\mathrm{re}) = \mathcal{D}_i\{x_i(n)\} = y_i(n) + s_{\mathrm{d},i}[y_i(n+1) - y_i(n)] \tag{25}$$

where $x_i(n)$ is the incoming signal from microphone i, and $y_i(n_\mathrm{re})$ is the re-sampled signal with a correct time delay with fractional samples $s_{\mathrm{d},i}$. This will then be performed for all microphones and added together according to Eq. (2). The power is calculated with Eq. (3).

3.6.4 Combination of delay methods

A combination of the previous mentioned time delay methods can be used. Both the sinc filtering and interpolation methods can delay a signal with both whole and fractional samples. But one can also first delay the signal with whole samples by padding, and then delay the remaining fractional sample with a sinc filter or interpolation.

4 System Architecture

4.1 Simulated Array

The simulated array is a separate part of the project that tries to mimic the array as good as possible on a FPGA. The idea is to test the sampling algorithm on the simulated array first to verify that it works properly. The simulated array can either be built on the same FPGA running the sampling or on an other FPGA that is connected together with jumper cables exactly like the real array. The simulated array sends a 8 bit mic id first and then a 16 bit counter in the 24 bit TDM slot. This makes it easy to see that data from all the microphones are received in the correct order and from the correct sample.

4.2 PL

The PL consists of the VHDL code to sample the microphone arrays and an AXI protocol to send the data over to the PS. The PL is built by different components and can be simplified as referred to figure 9.

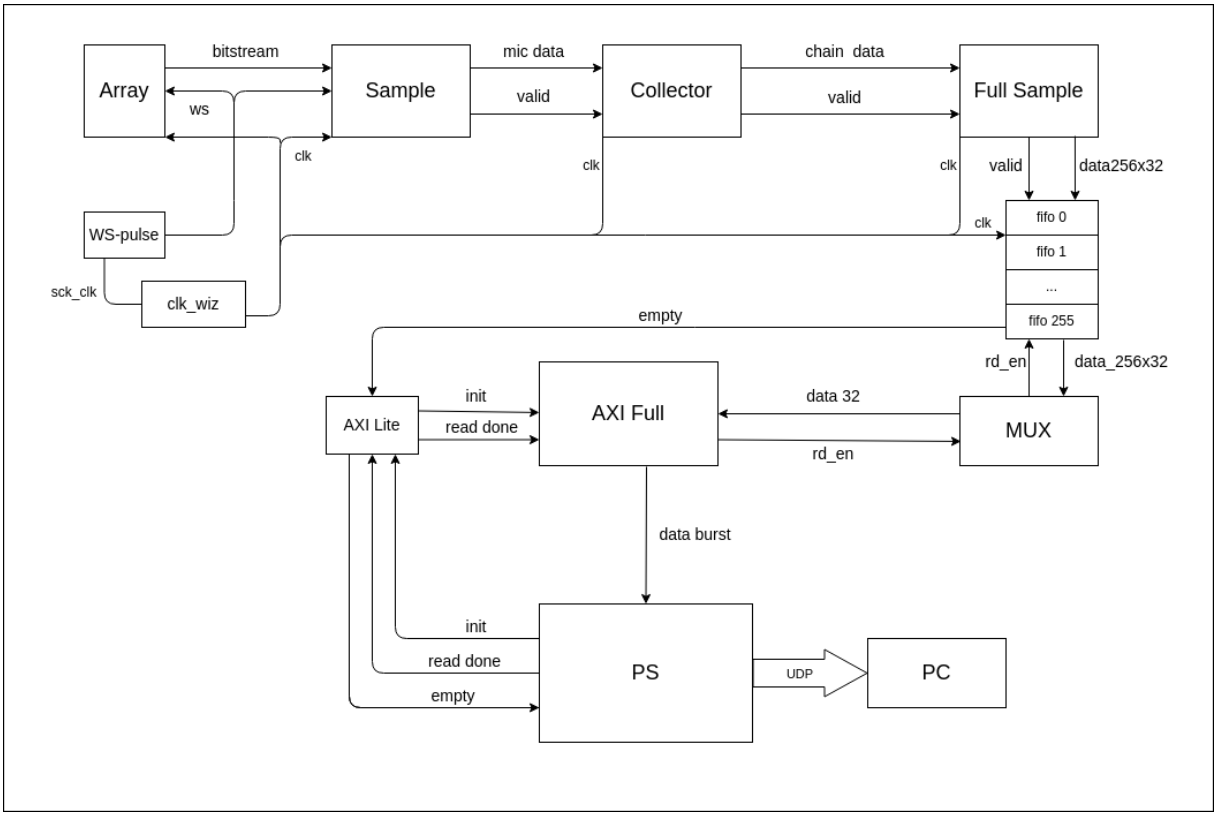


Figure 9: Structure of PL

4.2.1 Sample

```
entity sample is
   port (
      sys_clk              : in std_logic;
      reset                : in std_logic;
      bit_stream           : in std_logic;
      ws                   : in std_logic;
      mic_sample_data_out  : inout std_logic_vector(23 downto 0);
      mic_sample_valid_out : out std_logic := '0'
   );
```

```
end entity;
```

The sample component is the first logic handling the incoming bitstream from the microphone arrays. Each bit will be sampled at the frequency of the clk (125 MHz) after an incoming ws pulse. It runs on a state machine which controls the sampling process, ensuring synchronization with the system clock and of the ws signal to initiate sampling for each microphone in the chain. The state machine has three states: IDLE, RUN, and PAUSE.

The IDLE state waits for a rising edge of the ws signal to transition to the RUN state. It ensures that the sampling process starts at the same time that the microphone chain starts to send data.

The RUN state samples the incoming bit-stream from one microphone array and shifts each of the 24 bits into the mic_sample_data_out vector.

When one microphones have been sampled and the valid signal is triggered it enters the PAUSE state. This state waits for 8 additional clock cycles to allow empty TDM slots to pass before returning to the run state for the next microphone sample. This process is repeated for all 16 microphones in the chain. After all the microphones have been sampled the state machine returns to idle and waits for the next ws pulse.

A total of sixteen instances of the sample component is generated, one for each chain on the four microphone array.

4.2.2 Collector

```
entity collector is
   port (
      sys_clk               : in std_logic;
      reset                 : in std_logic;
      mic_id_sw             : in std_logic;
      mic_sample_data_in    : in std_logic_vector(23 downto 0);
      mic_sample_valid_in   : in std_logic;
      chain_matrix_data_out  : out matrix_16_32_type;
      chain_matrix_valid_out : out std_logic := '0'
   );
end collector;
```

The next step, the Collector, collects data from the each chain (16 microphones) of the sampled arrays and forms a 16x32 matrix of data (chain_matrix_data). Since the data from each sample only is 24 bits we will need to pad the data-width according to two's compliment to a 32 bit to ensure it fits the coming AXI4 Full protocol. chain_matrix_valid_out will signal the next component full sample that the data is ready to process. The Collector also have a switch input that enables debugging mode. In debugging mode the padding is changed to the number of the mic starting at 0 and going to 255.

4.2.3 Full Sample

```
entity full_sample is
   port (
      sys_clk                : in std_logic;
      reset                  : in std_logic;
      chain_x4_matrix_data_in : in matrix_16_16_32_type;
      chain_matrix_valid_in   : in std_logic_vector(15 downto 0);
```

```
      array_matrix_data_out   : out matrix_256_32_type;
      array_matrix_valid_out  : out std_logic;
      sample_counter_array    : out std_logic_vector(31 downto 0)
   );
end full_sample;
```

Full Sample will combine the data from the collector matrices into a larger matrix (array_matrix_data), and outputs a 256x32 matrix with data from all chains of four arrays, each sample being 32 bits. A valid signal array_matrix_valid_out is also output for the next block to start.

4.2.4 AXI4 Full

Once the data is collected and structured it needs to prepare for transfer from PL to PS and pass through a protocol, the AXI4 Full protocol, (Advanced eXtensible Interface protocol). The earlier version, AXI4 Lite, from last summer could not handle the data capacity alone of more than about 1,5 microphone array without lowering the sampling frequency, which would disturb the later beamforming algorithm. In short words it was in need of an upgrade if we wanted it to handle more data. A more in depth description of the components and the previous AXI4 Lite protocol can be found at [ND].

As per figure 9, after Full Sample the data matrix is passed through an synchronous FIFO (First-In-First-Out) memory which is capable of indicating empty and full status. Every microphone has its own FIFO, meaning that each write operation to the FIFO can input a 32-bit data value as wr_data. Once a rd_en signal for a read is received from the MUX/AXI Full protocol the tail pointer of the FIFO is updated. Similarly is the head pointer updates when a valid signal (write) is sent from Full Sample.

The 256x32 matrix is next passed through the MUX which will output a 32-bit vector of each microphone. AXI Full will send a signal rd_en which tells the MUX that it's ready to receive data. The MUX will start reading data from the FIFO matrix, with regards on its rising edges and pass through to AXI-Full. For easier debugging, of the MUX and the AXI, a switch can be used to set the MUX in debugging mode. Then the MUX will then send a predetermined bit combination instead of mic data.

The main part of the AXI protocol handles memory mapped burst write transactions of address/-data. It combines attributes of an AXI4 Full interface and an AXI4 Lite interface, as shown in figure 10. It features dedicated channels for various functions: Write Address, Write Data, Write Response, Read Address and Read Data channels. Transactions in AXI involve bursts of data transfers, and each transaction contains address, control, data, and response information, as seen in figure 11. [Inc]. In our application the data is transferred from PL to the PS side and the only data going in the other direction is status bits. Therefore the AXI Full will send bursts with a size of 256x32 from PL to PS and while the AXI-Lite sends and receive status for the transaction. See figure 10. When using AXI-Full in burst mode it will only need to handshake once per burst instead of once per transaction. To transfer the data bursts from PL to DDR3 memory on PS a high performance port (HP-port) is used to achieve a high throughput. By using incrementing mode on the AXI-Full the data in the burst will be placed on incrementing addresses in the DDR3 memory.

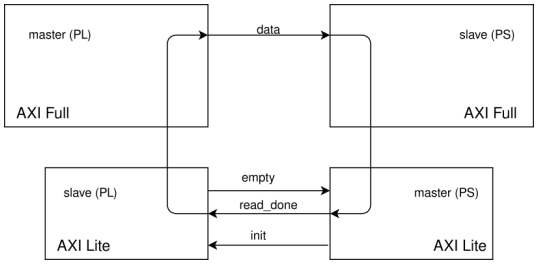


Figure 10: The relation between AXI4-Full and AXI4-Lite

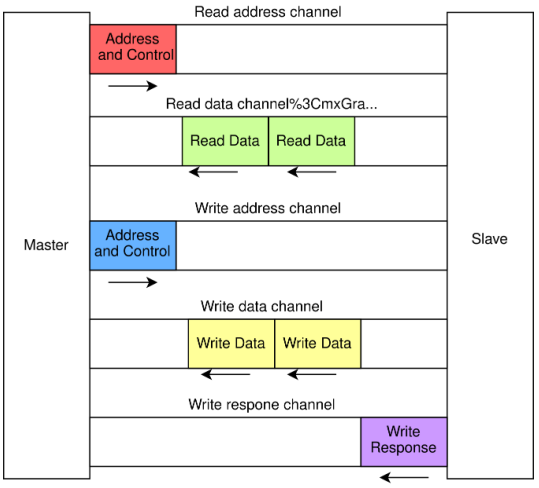


Figure 11: AXI Channels

4.3 PS

4.3.1 UDP Transmitter

The goal of the PS is to receive the mic-data from the PL side and the send it using UDP (User Datagram Protocol) over a Ethernet connection to a PC for real time processing. The first step is therefore to read the data that the AXI-Full have sent over to the DDR3 memory. The important thing is to use the correct starter address that which is set in the AXI-Full on the PL side and then incrementing by one for each new 32-bit data slot. When reaching the end of the 256 slots the D-Cache memory have to be flushed over the range to allow for new data to be received. When the processor is done reading the data it sends a read done signal over the AXI-Lite back to the PL side which allows for new data to be prepared. There is also a empty flag being sent from PL over AXI-Lite to indicate for the PS that it have to wait for new data to be sampled. After a full sample is received it is packet in to the UDP packet. The first two 32-bit slots of the packet is a payload header containing protocol version (8-bit), number of arrays (8-bit), and frequency information (16-bit), as well as a 32-bit sample counter, see figure 12. When everything is packed the network packet is ready it is placed in a buffer and the sent over Ethernet to the receiving PC.

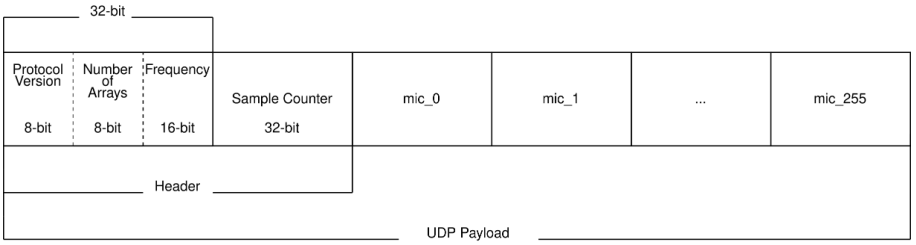


Figure 12: UDP Payload Protocol

4.4 PC

4.4.1 UDP receiver

To receive the samples sent from the Zybo there is a UDP packet receiver on the PC side written in C. This receiver creates and binds a socket on the port 21844. After a successful connection to the port it is possible to receive UDP messages. The receive function reads the header of the first incoming message and checks that the message is sent with the correct protocol and then writes the samples over to a buffer. This buffer is allocated as a shared memory object so that it is possible to receive and interpret the samples concurrently. When the bits arrive from the transmitter, they

are ordered after rows and columns. This conversion does not take into account the rotation of the single microphone array and must be handled further down in the chain.

4.4.2 CPU Bottlenecks

The Intel Core i5 CPU had 4 cores which prompted the use of multicore-processing for the beamforming and application. It was recognized that running the application in a non-RT operating system introduced latencies and interupts which further reduced the performance of the program.

To combat the issue and to allow for faster computation, some calculations like delay in the time-domain were executed using AVX256 instructions for a higher throughput using the Single instruction, multiple data (SIMD) model for vectorized operations. The operation fused-multiply-add (FMA). This allowed for the CPU to use specialized registers for the floating-point arithmetic used in the summation of the different signals the in the beamforming algorithms.

$$a \leftarrow a + (b \times c) \tag{26}$$

26 was used for the delay using convolution with a sinc filter 22

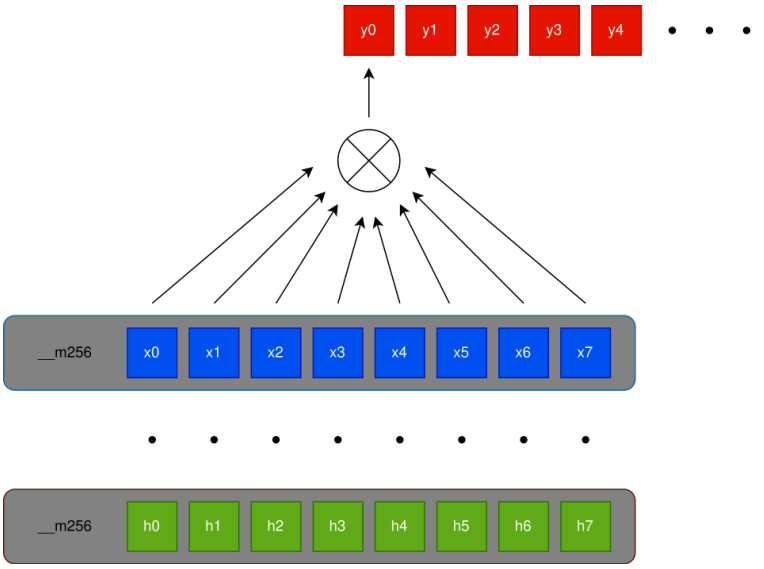


Figure 13: FMA

4.4.3 Cython wrapper

The API works in the following fashion

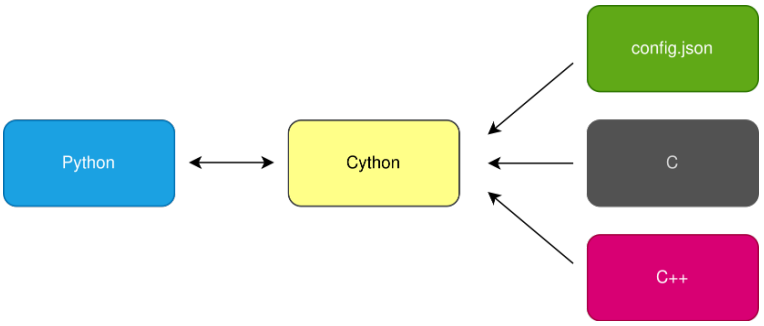


Figure 14: API language flow

The program exposes a set of methods that can be used in python in the following way: e. (can be fixed by turning duplicated AP on and off giving STA some time to reconnect)

The Beamformer application uses a ringbuffer to keep track of the received data

4.4.4 Django web-application

For simple use of all backend functionality a web application was created using the Python framework Django. The application displays the feed from the camera and has buttons for enabling the different beamforming backends. Once a backend is enabled the heatmap is drawn on top of the camera feed. In this mode it is also possible to adjust the threshold for which amplitudes the heatmap should be drawn using two sliders. It is also possible to disable the beamforming backend and return to the initial page with only camera feed where another backend can be selected.

5 Results

5.1 Microphone signals

A 2 s recording of a sinus signal of 1 kHz have been analyzed. The recorded signals are presented individually in figure 15. In this figure only three periods have been plotted, and microphones could possibly sent incorrect values might not be shown in this figure. Signals from selected microphones have been plotted in figure **??**.

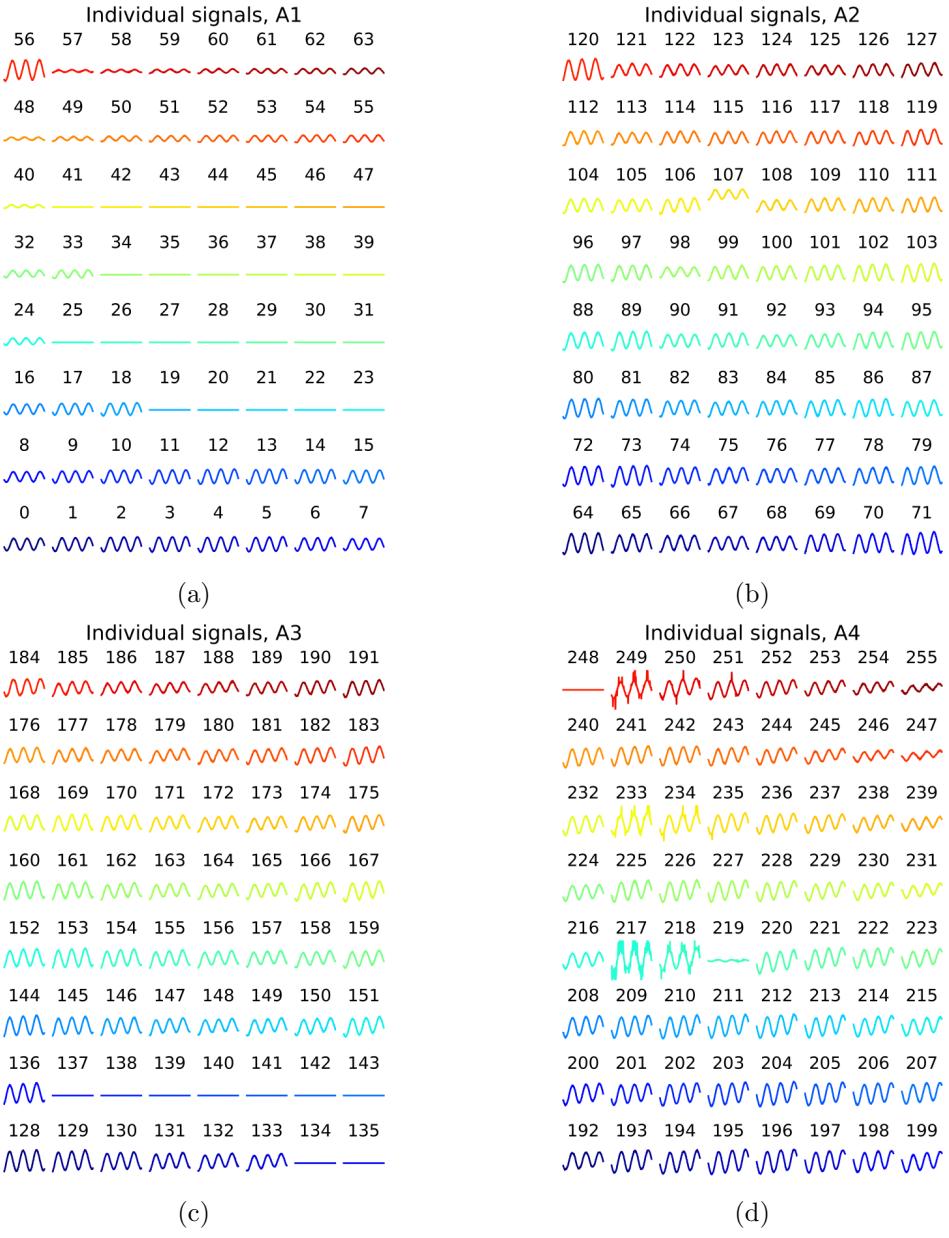
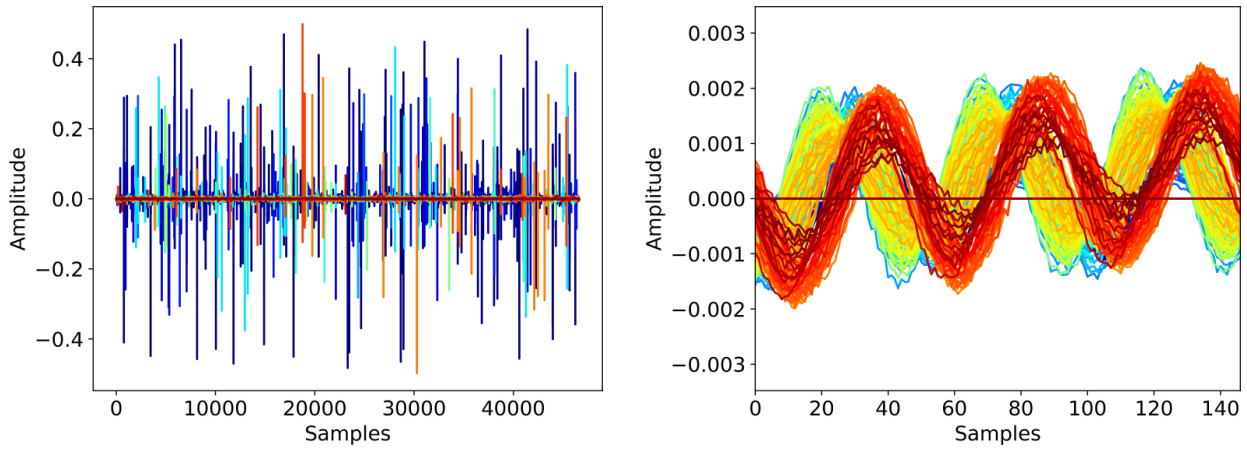


Figure 15: Three periods of the signals from the microphones in the three arrays. (a) Array 1 (PCBA2 DEFECTIVE), (b) Array 2 (PCBA1), (c) Array 3 (PCBA2), and (d) Array 4 (version 1).



(a) (b)

Figure 16: (a) All bad signals from all arrays, and (b) all good signals from all arrays when the bad signals in figure (a) have been deleted.

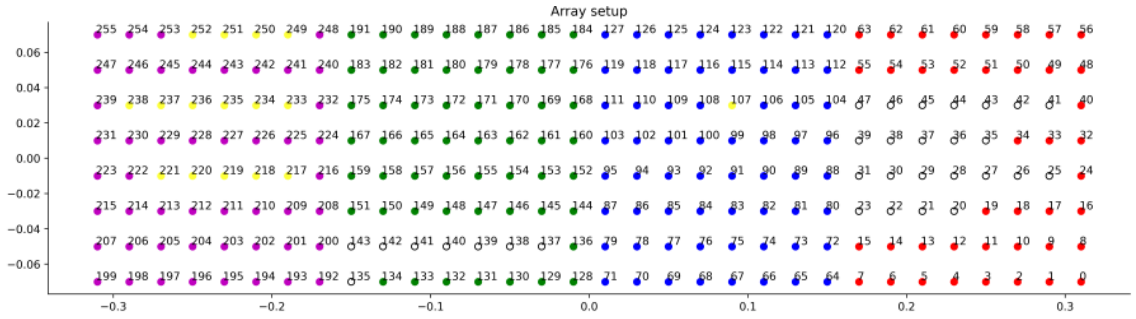


Figure 17: Array with bad microphones marked with a yellow dot, and microphones sending zero is unfilled.

5.2 Beamforming performance

The performance of the beamforming algorithms was first tested on emulated data that was generated in the same way as described in [ÅS]. When defining the sources used for generating the emulated data, we were able to choose the following settings:

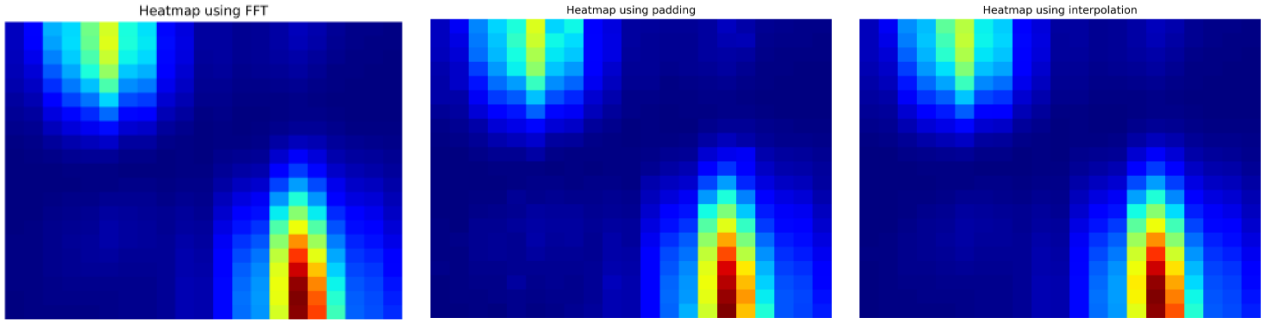
- Number of sources (1 or 2).
- Distance from array.
- Start and end frequency of sinus signal.
- Source angle θ.
- Source angle φ.

Where each setting could be set individually for each source (except number of sources).

The setup used for the array configuration was the same as described in section 3.2, and tests were conducted using one array and three arrays in the configuration. The reason for not performing any simulations with four arrays, was that we only had three working arrays until two days before the demo. The view angle, α, used in all the following simulations was 68°. When simulations were made with emulated data, it was assumed that all microphones in the array configuration behaved the same and worked perfectly. The signals were assumed to be perfectly sampled at the same time by all microphones. It was also assumed that the surroundings of the array and sources were perfect, i.e. no reflections, no interference between the PCB in the array and the emitted sound, and so forth.

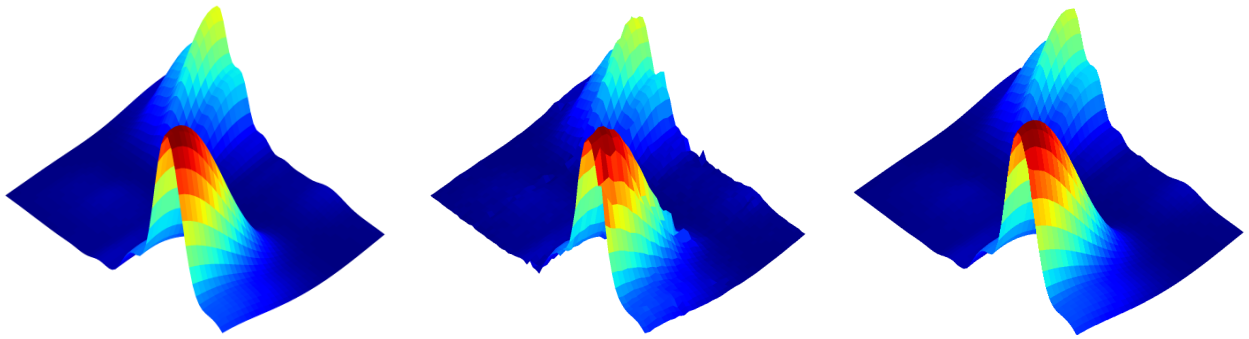
5.2.1 Comparison of the beamforming algorithms with emulated data

When testing the performance of the beamforming algorithms, we realized that the method of convolving with a sinc filter was unnecessary since it performs very similarly to the other time-domain methods but slower, while still being faster than the FFT implementation. We thus chose to only further investigate the three other methods.



(a) (b) (c)

Figure 18: Heatmaps illustrating the results from the beamforming algorithm when using FFT (a), padding (b) and interpolation (c).



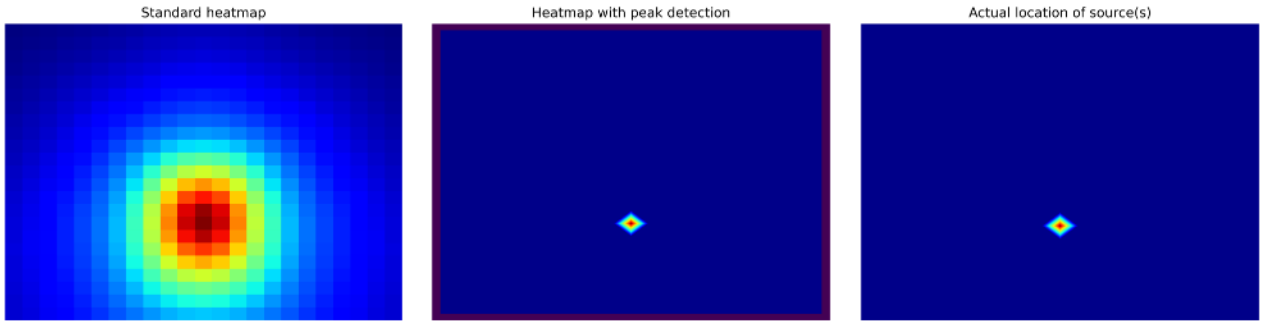
(a) (b) (c)

Figure 19: 3D illustrations of the resulting heatmaps, for a high resolution of 70 scanning points in x-direction and 35 in y-direction. With the beam forming algorithm version of FFT (a), pad (b), and interpolation (c).

In Fig. 18, a resolution of 21 scanning points in both the x- and y-direction was used. Here we can see that the results of the different beamforming methods are almost identical. However, looking at the 3D plots in Fig. 19, we see more noticeable differences as the resolution is increased. We see that the padding in Fig. 19b has a less smooth surface as it can't perform the phase shift with as high resolution as the other methods.

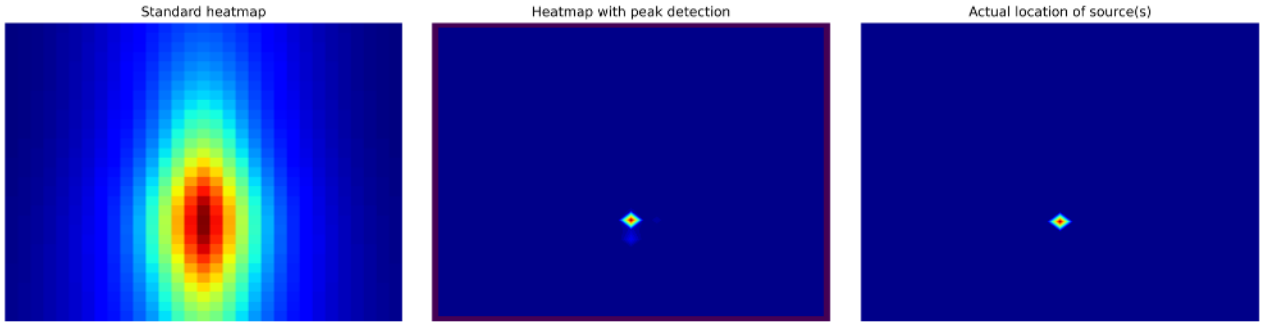
5.2.2 FFT results using emulated data

Results of simulations using the FFT algorithm are shown in the following figures. More emulated data results can also be found in the appendix.



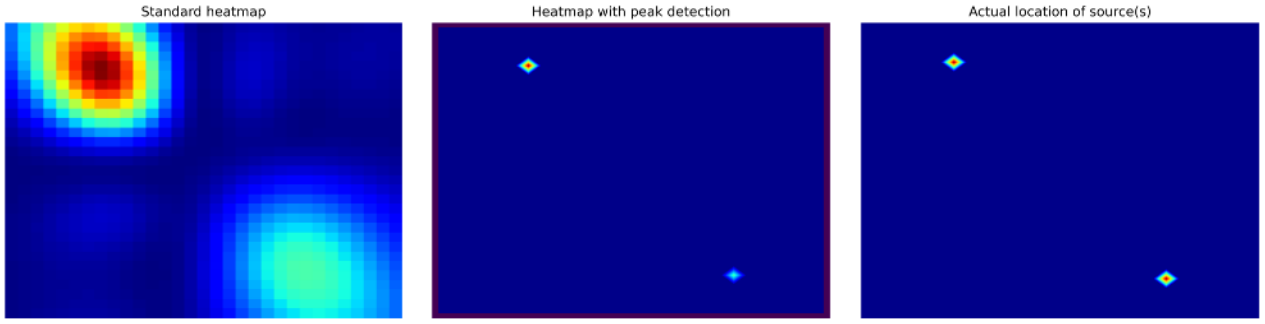
(a) (b) (c)

Figure 20: Simulation using one array and one source at: frequency band $f = [200, 8000]$ Hz , $\theta = 10°$, $\varphi = -90°$.



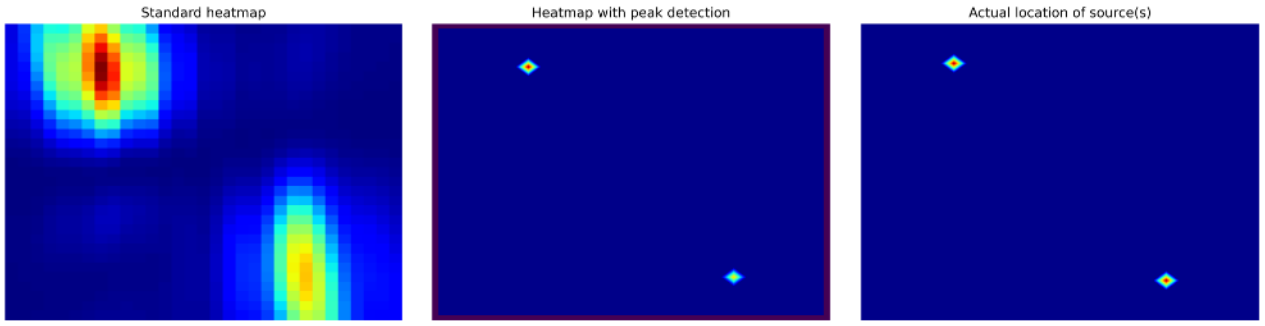
(a) (b) (c)

Figure 21: Simulation using three arrays and one source at: frequency band $f = [200, 8000]$ Hz, $\theta = 10°$, $\varphi = -90°$.



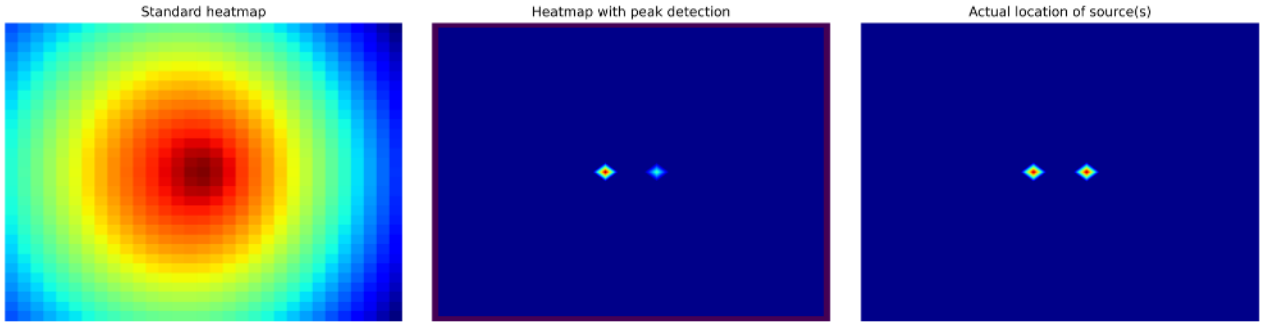
(a) (b) (c)

Figure 22: Simulation using one array and two sources at: $f_1 = [4000, 5000]$ Hz, $\theta_1 = 27°$, $\varphi_1 = -45°$ and $f_2 = [6000, 7000]$ Hz, $\theta_2 = 27°$, $\varphi_2 = 135°$.



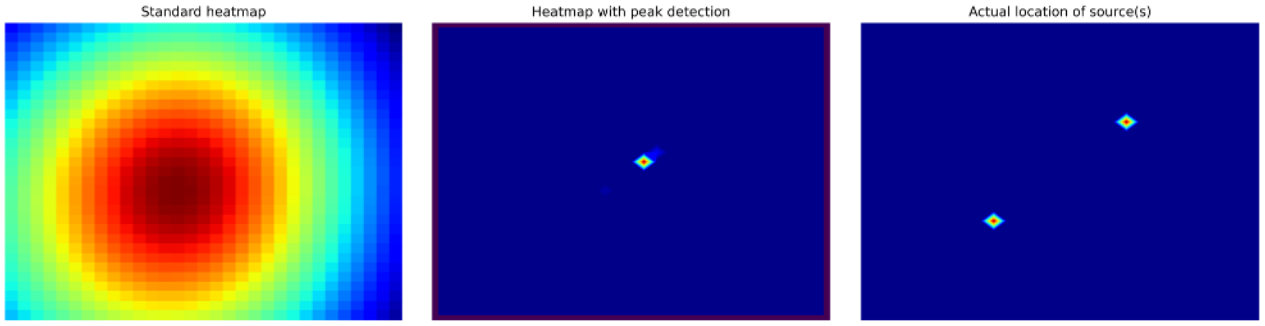
(a) (b) (c)

Figure 23: Simulation using three arrays and two sources at: $f_1 = [4000, 5000]$ Hz, $\theta_1 = 27°$, $\varphi_1 = -45°$ and $f_2 = [6000, 7000]$ Hz, $\theta_2 = 27°$, $\varphi_2 = 135°$.



(a) (b) (c)

Figure 24: Simulation using one array and two sources at: $f_1 = 200$ Hz, $\theta_1 = 5°$, $\varphi_1 = 0°$ and $f_2 = 300$ Hz, $\theta_2 = -5°$, $\varphi_2 = 0°$.



(a) (b) (c)

Figure 25: Simulation using one array and two sources at: $f_1 = [300, 1000]$ Hz, $\theta_1 = 15°$, $\varphi_1 = 35°$ and $f_2 = [400, 2000]$ Hz, $\theta_2 = -15°$, $\varphi_2 = 35°$.

5.2.3 Comments on results from FFT using emulated data

If we compare Figs. 20 and 21, we see that the beamwidth is reduced in the horizontal direction when using more arrays, because the arrays are stacked horizontally. However, we see that the number of arrays doesn't really affect the peak detection as it is able to accurately pick out the source location in both cases. Looking at Figs. 22 and 23, we see that both the standard heatmap algorithm and peak detection is able to pick out both sources as they emit at different frequency bands. If these sources were to emit at lower frequencies, as shown in Fig. 44 in the appendix, we see that the standard heatmap algorithm is unable to distinguish the two sources. As discussed in section 3.5.4, the two

beams targeting the sources start to overlap, which creates a beam/peak in between them. In Fig. 44b however, we see that the peak detection is still able quite accurately find both sources. In Fig. 24, we also see that the peak detection works well even when the sources are close together, whereas it is impossible to tell exactly where each individual source is located on the standard heatmap. In Fig. 25, we see that the peak detection fails as the sources emit at overlapping frequency ranges, which is expected according what was discussed in section 3.5.4.

When comparing Figs. 40, 41, 42 and 43 in the appendix, we see that the beamwidth on the standard heatmaps is reduced as the frequency is increased, which is expected. Looking closer at the higher frequencies in Figs. 42a and 43a, we see more noticeable side lobes, which is also expected since they start to move in to the width angle of the scanning window at higher frequencies.

Looking at the results from the simulations made using adaptive configurations in Figs. 47, 48, 49, 50, and comparing them to the standard algorithm results shown in section 5.2.2, the two methods perform similarly when it comes to finding and visualizing sources. However, as explained in section 3.3.2, adaptive configuration could be used to reduce computational time.

5.2.4 Results using FFT on recorded data

The beamforming algorithm was also tested on recorded data, where three different tests were conducted. In the first test, a 1 kHz sine-signal was played from a phone. The phone was placed approximately 1 m away from the microphone array. It was aligned with the center of the array in the horizontal direction, and was placed at a height just below the bottom of the array. In the second test, a 4 kHz sine-signal was played from the exact same location. In the third test, two phones were used to play static noise simultaneously. The phones were placed 1 m away from the array, at each end of the scanning window in the horizontal direction. They were also placed at the same height as in the previous tests. Three arrays were used in the configuration for these tests. Contrary to the assumptions made when simulating with emulated data, it was known that some microphones were bad. It was therefore interesting to visualize the recorded data results when using all microphones and compare them with results where the bad microphones were excluded.

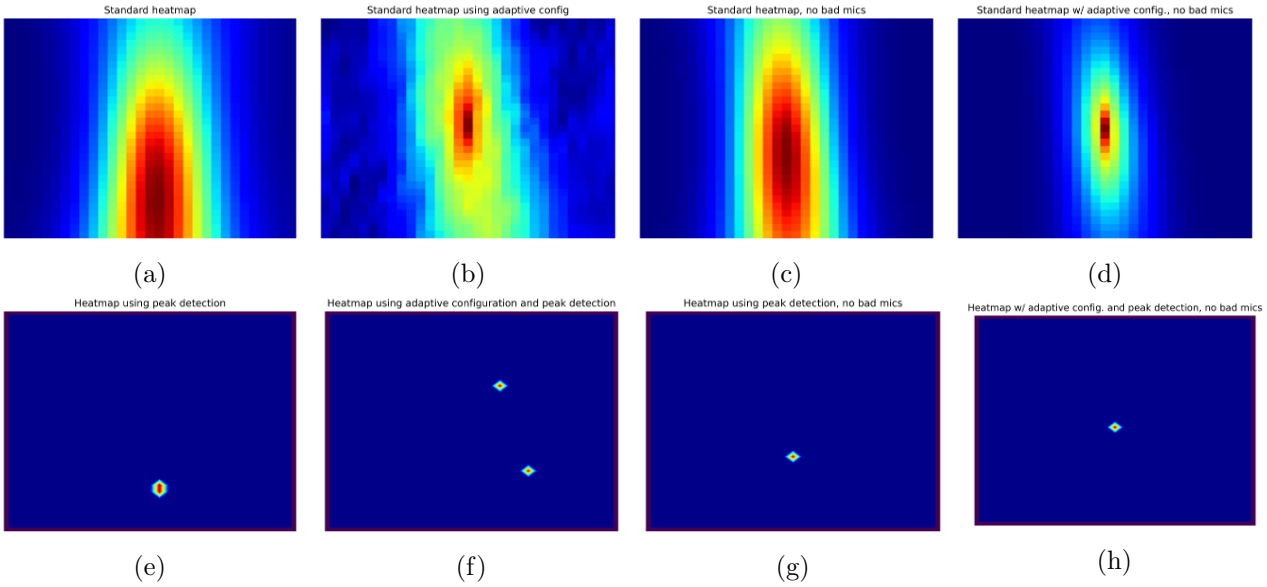


Figure 26: Results of recorded 1 kHz sine-signal.

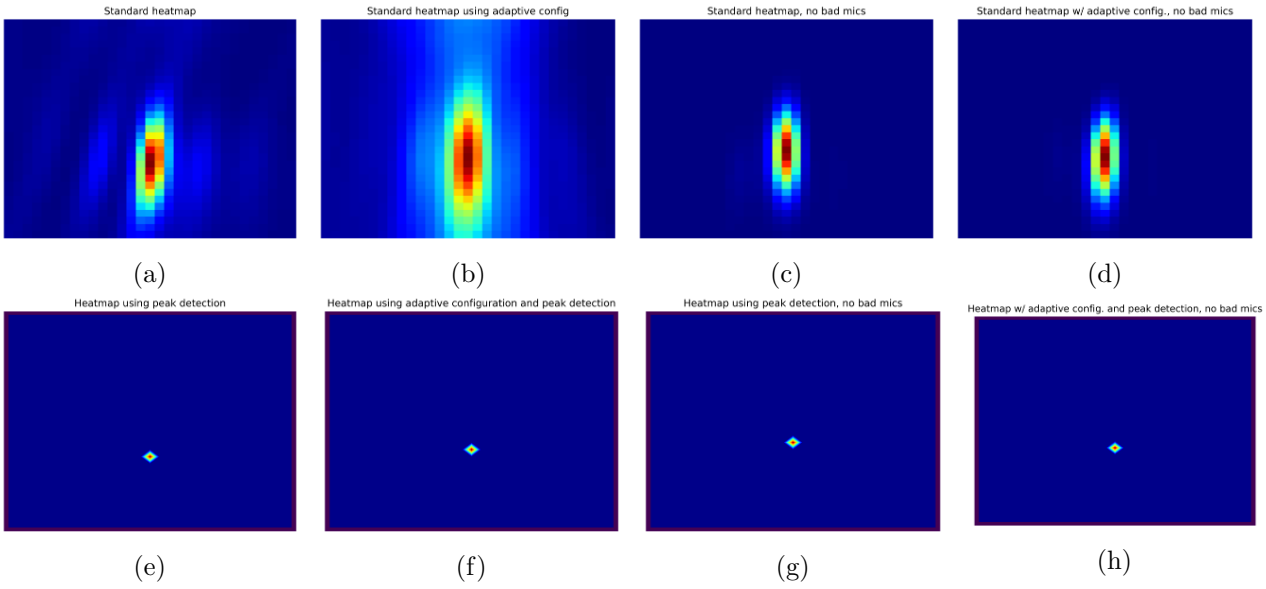


Figure 27: Results of recorded 4 kHz sine-signal.

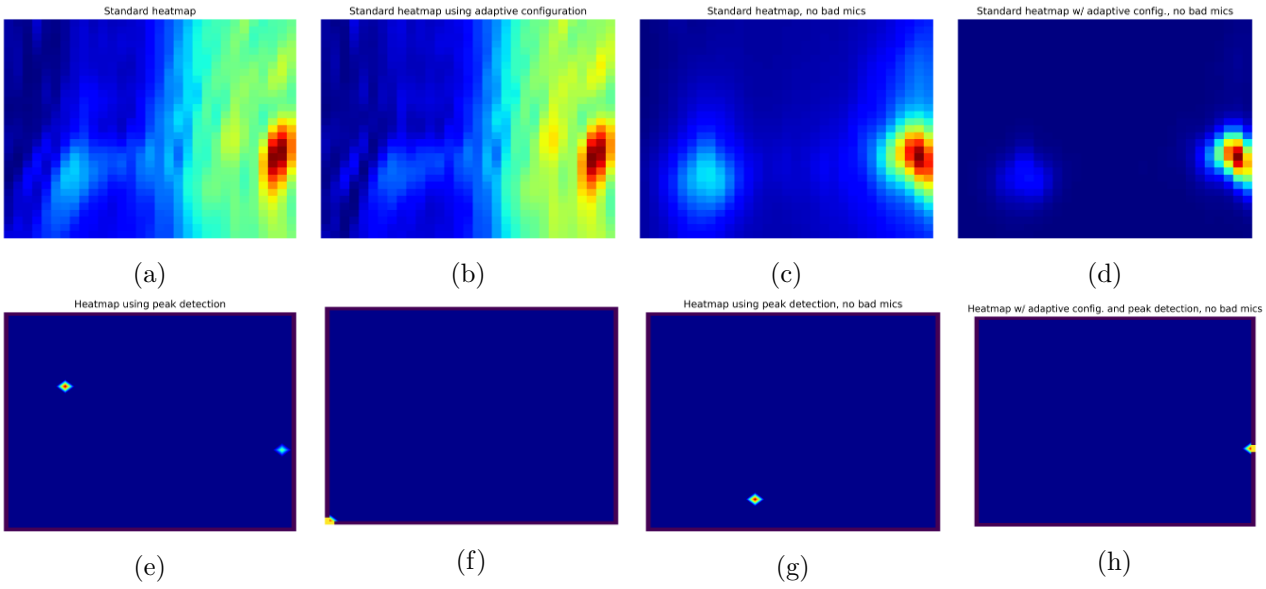


Figure 28: Results of two recorded static signals.

5.2.5 Comments on results from FFT using recorded data

In Fig. 26, we see some mixed results of the beamforming on the 1 kHz sine-signal. All results show that there is a source in the middle of the scanning window, in the horizontal direction. The height of the source however, varies between the results. The beamwidth is quite wide if adaptive configuration is not used, which is understandable since 1 kHz is a relatively low frequency for the default array configuration. In Fig. 26f, we see that the peak detection when using adaptive configuration and all microphones was bad in this test. This could be due to that one or several microphones used in the adaptive configuration is bad. The negative effects of having bad microphones are greater when using adaptive configuration, since there are less microphones used. In Fig. 26h, we see that the peak detection is much better when all bad microphones are excluded from the adaptive configuration, although it might not be a perfect result since we can't confirm exactly where the source was located on the scanning window.

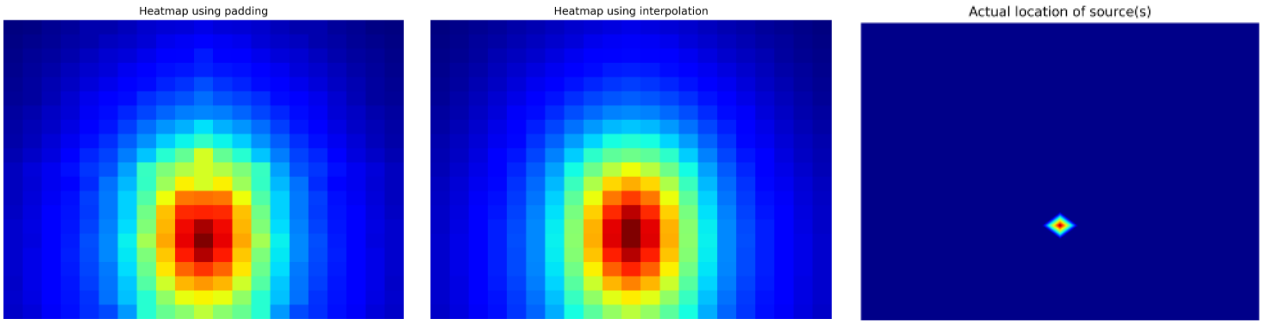
Looking at the results from the 4 kHz sine-signal in Fig. 27, they all agree with each other. Which

may indicate that the microphone array handles signals around that frequency well, if the results are accurate.

In Fig. 28, when looking at the standard heatmaps, we see a difference in performance when using all microphones vs. excluding the bad microphones. When all microphones are used, one source is very highlighted and the other is barely visible. When the bad microphones have been excluded, both sources can be seen and are distinguishable from each other, but the peak at the left source is much weaker than the right source with adaptive configuration. However, the peak detection performs quite poorly in all cases, either showing incorrect source locations or only showing the approximate location of one source. But it is expected that the peak detection does not show great results since there are two sources emitting at the same frequency.

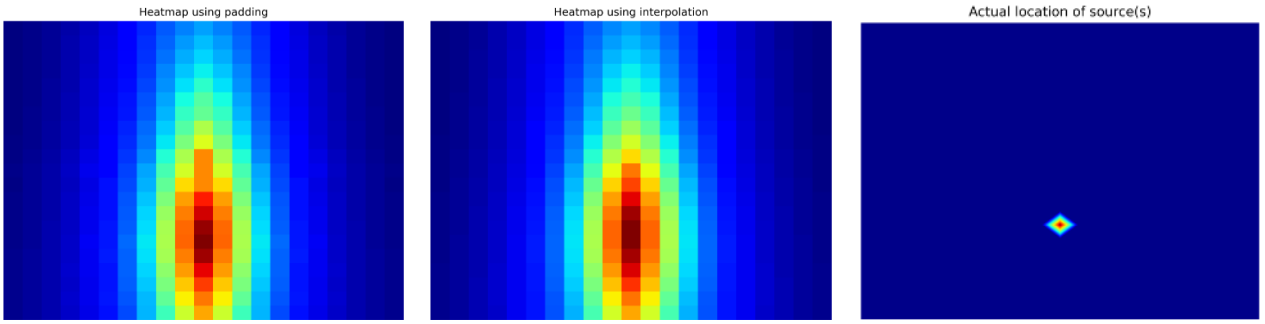
5.2.6 Time domain results using emulated data

When testing the time domain beamforming algorithms with emulated data, we defined the sources the same way as for the FFT. Most of the results from the time domain algorithms were very similar or near identical to each other, and to the FFT results. We therefore chose to show a few of them in this report. We did not have time to develop a fully functioning peak detection algorithm in the time domain, which is something that may be a future implementation/improvement. Results of simulations using both the padding- and interpolation algorithm are shown in the following figures.



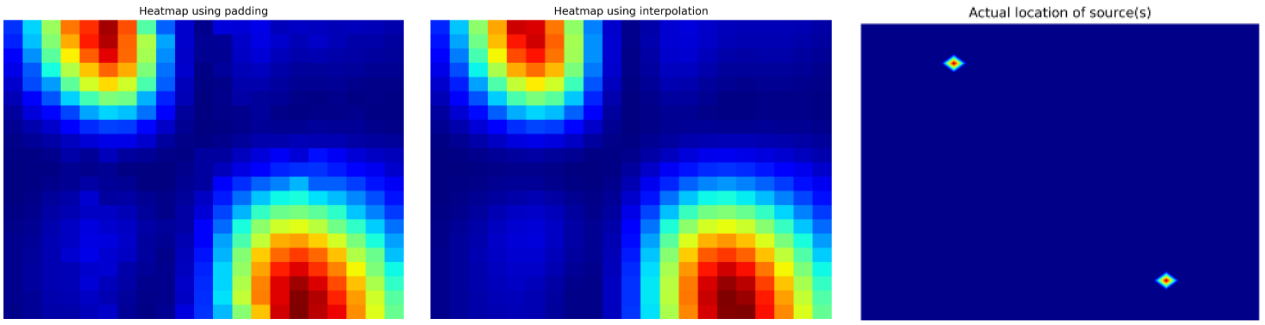
(a) (b) (c)

Figure 29: Simulation using one array and one source at: frequency band $f = [200, 8000]$ Hz, $\theta = 10°$, $\varphi = -90°$.



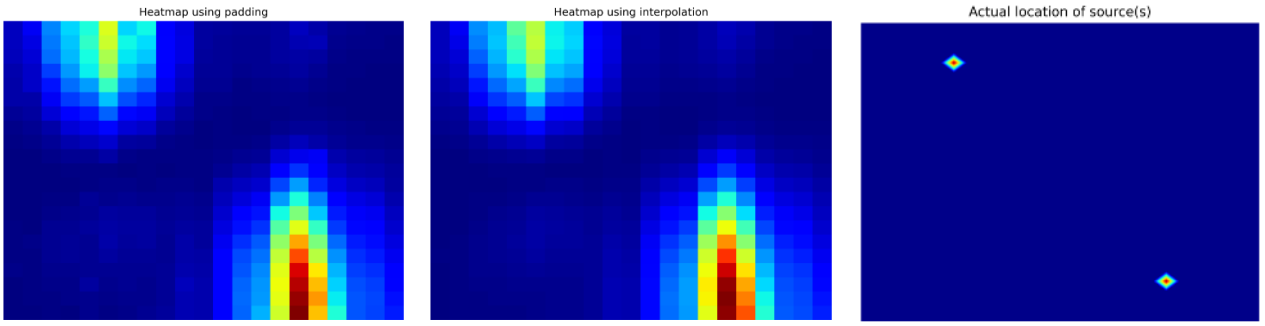
(a) (b) (c)

Figure 30: Simulation using three arrays and one source at: frequency band $f = [200, 8000]$ Hz, $\theta = 10°$, $\varphi = -90°$.



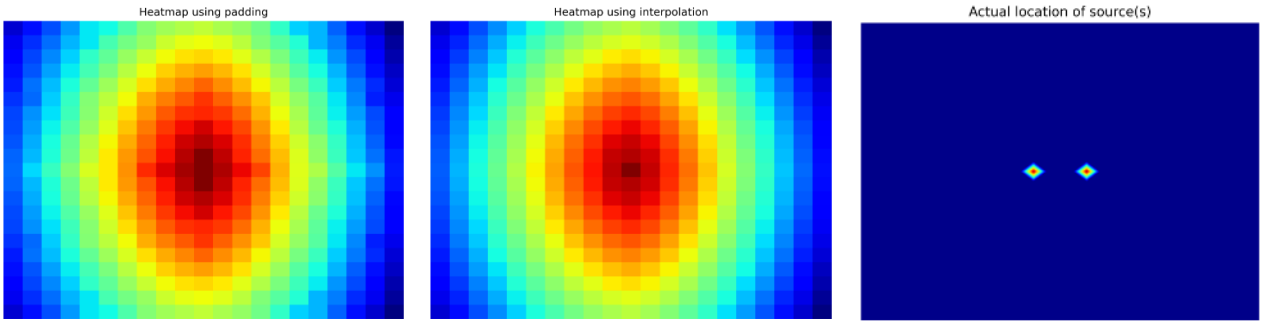
(a) (b) (c)

Figure 31: Simulation using one array and two sources at $f_1 = [4000, 5000]$ Hz, $\theta_1 = 27°$, $\varphi_1 = -45°$, $f_2 = [6000, 7000]$ Hz, $\theta_2 = 27°$, $\varphi_2 = 135°$.



(a) (b) (c)

Figure 32: Simulation using three arrays and two sources at $f_1 = [4000, 5000]$ Hz, $\theta_1 = 27°$, $\varphi_1 = -45°$, $f_2 = [6000, 7000]$ Hz, $\theta_2 = 27°$, $\varphi_2 = 135°$.



(a) (b) (c)

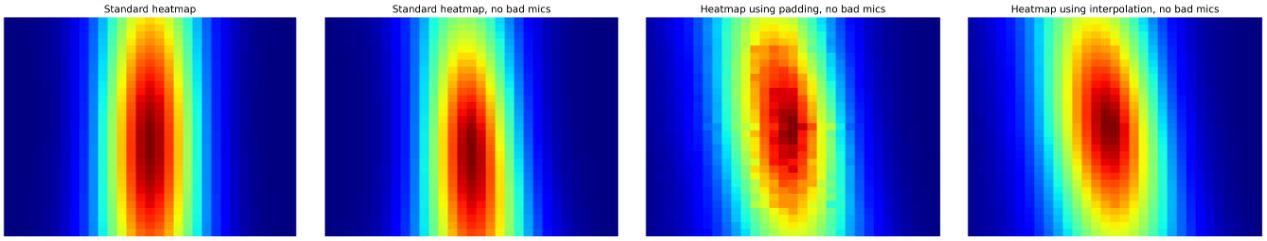
Figure 33: Simulation using one array and two sources at: $f_1 = 200$ Hz, $\theta_1 = 5°$, $\varphi_1 = 0°$ and $f_2 = 300$ Hz, $\theta_2 = -5°$, $\varphi_2 = 0°$.

5.2.7 Comments on results from time domain algorithms using emulated data

When looking at the results we see that both algorithms are good at handling singular sources at a wide range of frequencies. They can also handle multiple sources as long as they aren't too close to each other and emit at too low frequencies, causing wide beams and potential overlapping, which can be seen in Fig. 33. When comparing these results with the FFT results, we see that that the output of the time domain algorithms generally is the same as the FFT algorithm's. For our implementation, the results show that the considerable differences between the algorithms are the computational speed and the picture quality of the heatmap when the resolution is increased.

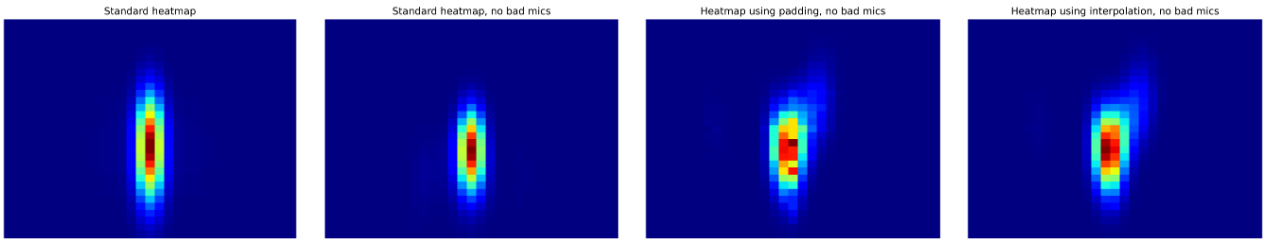
5.2.8 Comparison between emulated and recorded data results

The following figures compare the results of emulated 1 kHz and 4 kHz sine-signals placed approximately at the same scanning window location as the recorded sine-signals to see how well they matched. The results are taken from a single frame in the recorded data.



(a) (b) (c) (d)

Figure 34: Heatmaps showing result of emulated 1 kHz sine-signal when using FFT (a), and recorded 1 kHz sine-signal when using FFT (b), padding (c), and interpolation (d).



(a) (b) (c) (d)

Figure 35: Heatmaps showing result of emulated 4 kHz sine-signal when using FFT (a), and recorded 4 kHz sine-signal when using FFT (b), padding (c), and interpolation (d).

In Figs. 34 and 35, we see that the results for the emulated and recorded data matches quite closely for all methods. Note that the bad microphones have been excluded in the recorded data results whilst all microphones have been used for the emulated data, which might be a reason for the differences between the emulated and recorded results. Considering how similar the results are, we are confident that the beamforming algorithms and microphone array are able to handle real data pretty well when all bad microphones are excluded.

5.3 Real-time results

The algorithm run in real time performs slightly different compared to what have been presented as results on recorded data in section 5.2.8. For one sound source, the beamformer can find the direction of the source with high accuracy. When two sources of drone sound are present, with approximately the same amplitude, the beamformer often fails to visualize both sources in the same frame, as seen in figure 36. Here we see three frames taken after each other, where only one of the frames show both sources clearly. This is most likely due to how the post processing of the image, which is more sensitive to the amplitude of the sources compared to the human ear.

We also performed tests when flying with a real drone. In figure 37 we have frames from when the drone flies relatively close to the array. Here we can see a clear detection of the drone. However, it has an offset in figure 37a because the camera was not pointed exactly in the right direction. Figure 37b is a frame from almost directly after when the drone is flying fast to the left, showing that we have a delay in the beamforming.

When flying with the drone in the same location, but much further away, we can see the effects

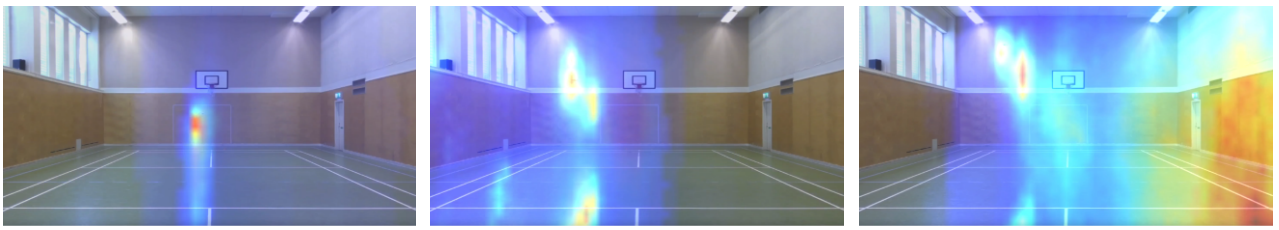
of reflections from the walls and floor. In figure 38a the beamformer can accurately find the drone. The reflections from the floor is presented in figure 38b, and reflections from the wall is clear in figure 38c.

(a) (b) (c)

Figure 36: Real time beamforming when two sources of drone sound is present, showing frames taken after each other.

(a) (b)

Figure 37: Real time beamforming with one sound source being a drone, flying close to the array. Frames are taken after each other with a short time gap.



(a) (b) (c)

Figure 38: Real time beamforming with one sound source being a drone, flying close to the array. Frames are taken at different times during the flight.

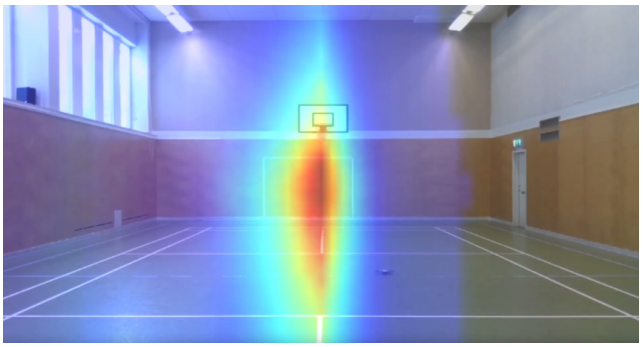


Figure 39: Resulting heatmap when no sound source is active, and only background noise is visualized.

5.4 Spatial filtering in one direction

Some undocumented tests where performed to spatially filter out sound from one desired direction. This was done by placing two mobile phones with some separation between each other, and letting them play two different songs at the same time. Then, it was possible to listen to the filtered spatially filtered signal in real time via headphones. We could then distinguish one song rather clearly when pointing the beam in this direction, while the other song had been noticeably suppressed. No qualitative tests, apart from the stated, of the performance on spatial filtering was conducted. This was due to limited time and test capacity.

6 Future Work

6.1 Beamforming algorithm

6.1.1 Microphone directionality

The microphones are stated to have an omni-directional response. However, it seems like the directionality of the microphones are changed when mounted to the PCB. This increases the sensitivity to sound arriving from the forward direction, and decrease sensitivity from the sides.

6.1.2 Direction of arrival algorithms

A very relevant future improvement of the beamforming algorithm is to estimate the of direction of arrival with a higher resolution than the delay and sum beamforming. This could both be of interest for the visualization of the sources in the heatmap, but also for indentification and tracking of moving source. An implementation of such beamforming algorithms could for be MVDR, or other DOA algorithms than can be applied on or modified to wide-band signals.

6.1.3 Tapering window

No tapering window was used in the beamforming algorithms. This was not considered since the side lobe levels in the most used frequency range of the possible sound sources (¡8 kHz) was very low, see Fig. 4, if all microphones where active. However, if we only use every other microphone, mode 2, or less it could be of interest to reduce the side-lobe-levels with tapering of the elements.

Additionally, an introduction of a tapering window could possibly help to reduce the significant amplitude increase from the forward direction due to the non-omnidirectional microphone response pattern.

6.1.4 Padding

Instead of delay the signals with padding with 0, as described in section 3.6.1, we could pad with the values from the previous frame. Then, we will not lose parts of the signal which this might solve the clipping when listening to the beamformed sound. Additionally, larger angles of arrival require a greater time delay and the signal will be padded with more zeros. Then, the calculated power for larger angels will be lower compared to the forward direction where no time delay and padding is required. This could also partly explain why we in the real time run obtain a lobe in the middle of heat-map, see figure 39. Hence noise from forward direction will have higher power than from other directions, and this could be avoided if we pad with previous values instead of zeros.

A solution is to use a rolling buffer and keep track of the N latest samples. This might allow for smoother transition between frames.

6.1.5 Peak detection

Getting good results using the peak detection algorithm is completely reliant on having sources that emit at different frequencies and choosing threshold values that are optimal for the setting and surroundings of the array. Because of this, its good to use for very specific situations but not very dynamic in that sense. To make it more useful, the peak detection could possibly be combined with algorithms such as MUSIC and MVDR to make it easier to detect and distinguish peaks.

6.2 Runtime config

6.3 UDP

6.3.1 UDP handshake

The receiver tunes in to the UDP datagram stream, but it has no way of controlling or see how the packages are structured, therefore in order to receive the correct message size, the number of arrays were defined (hard-coded) on both the FPGA and the PC. This is a wasteful operation if not all arrays are connected or required.

This could be implemented with a simple handshake between the PC and the FPGA which might also allow for controlling the FPGA during runtime. This will also reduce the total packet size, but that might be marginal since only 64 bytes are used for header data for each sample.

6.3.2 UDP playback

Wireshark was initially used to replay the receive messages, but it would be beneficial to write a little program that can replay the packets in realtime for debugging purposes.

6.4 Sampling

6.4.1 Sampling counter

The sample counter is created in full_sample, but when the protocol changed from AXI-Lite to AXI-Full it was never updated and instead set as a constant. IAn implementation of this feature would be beneficial for debugging purposes. A possible solution would be to send the sample counter from full_sample to an AXI-Lite register that is read from PS. For this to be possible there needs to be a FIFO for the sample counter similar to the FIFO's currently in use for the data to have the counter in sync with the samples.

6.4.2 Simulated array

The simulated array was designed to run on the same FPGA as the sampler as well as on a separate FPGA connected together with jumper cables, as described in section 4.1 Currently, the simulated array have to be fixed for both applications. The simulated array that runs on the same FPGA have the wrong timing since it had to be changed late when longer cables where introduced. Then the second problem is that the simulated array that runs on a separate FPGA have problems with the clocks making us miss some samples. This is a harder problem that might take a bit of work to fix.

6.4.3 Simulated array order

This is just a small fix for the simulated array on the same FPGA as the sampler when we are running with four arrays that are not in the order of the connections (JE fisrt, JB second, JC third and JD last) it would be nice to also have the simulated array able to work in this order. This could simply be done by swapping some outputs in the simulated_array.vhd. It would be nice to have a parameter to set this from the top file.

6.4.4 Automatic delay calibration

There is currently an index created for each sample.vhd in the top file. This is a index for the delay length and it would be nice to have some sort of automatic setup of this number. This could be done by examining the most significant bit (MSB) and verifying that it matches the following 7 bits; if they do not match, the sample delay is not accurately calibrated. An automatic check for this that changes this index on the fly would be nice to reduce the amount of calibration needed.

6.4.5 Communication from PC to FPGA

To reduce the need for rebuilding the VHDL code some future work would be for the PC to be able to set some parameters on the FPGA. This could be done by sending UDP packets from the PC to the FPGA and then sending them down though AXI-Lite. This would allow the user to change the recording frequency, delays, debugging modes and more.

6.4.6 Sampling and beamforming

Some future work would be to implement parts of or all of the beam-forming on the FPGA and the Zynq processor. If possible it would make a small and convenient package where both the sampling and beam-forming is done on the same development board. There might be a need for sending data back and forth from the FPGA and the ZYNQ to complete this feature but it should be possible. The goal partly being to utilize the FPGA combined with both cores on the Zynq for maximum performance.

Bibliography

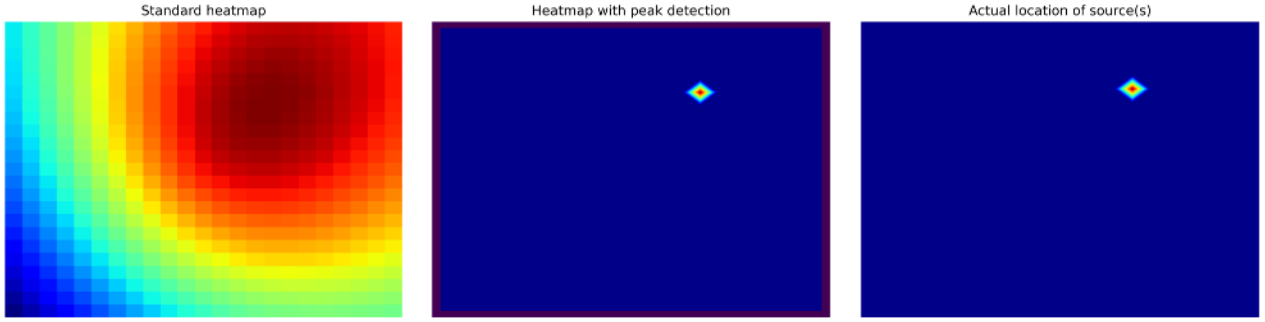
[ÅS] Lucas Åkerstedt and Mika Söderström. *Phased microphone planar arrays*. URL: https://github.com/acoustic-warfare/micarray/blob/main/Signal_processing_AW.pdf. (accessed: 13.07.2023).

[Inc] Xilinx Inc. *Xilinx, UG761 AXI Reference Guide*. URL: https://docs.xilinx.com/v/u/en-US/ug761_axi_reference_guide. (accessed: 07.08.2023).

[ND] Ivar Nilsson and Jacob Drotz. *Acoustic-Warfare: FPGA sampling*. URL: https://github.com/acoustic-warfare/FPGA-sampling/blob/main/doc/acoustic_warfare_fpga_sampling.pdf. (accessed: 13.07.2023).

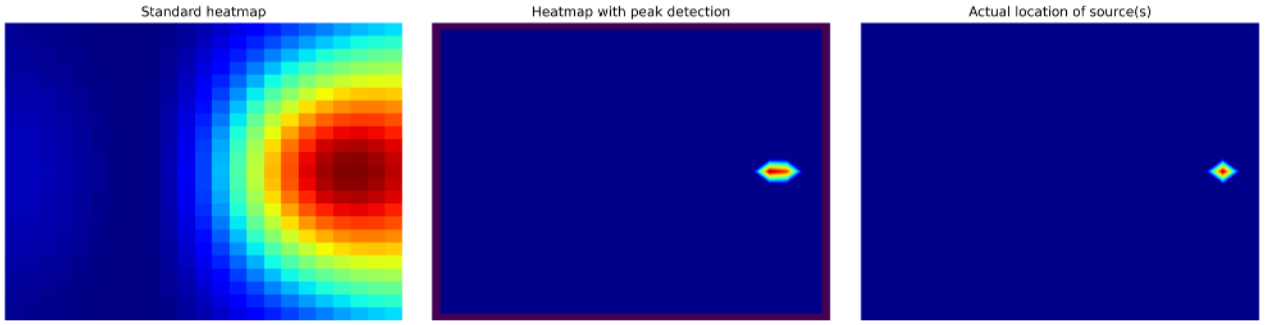
Appendix

Emulated data results



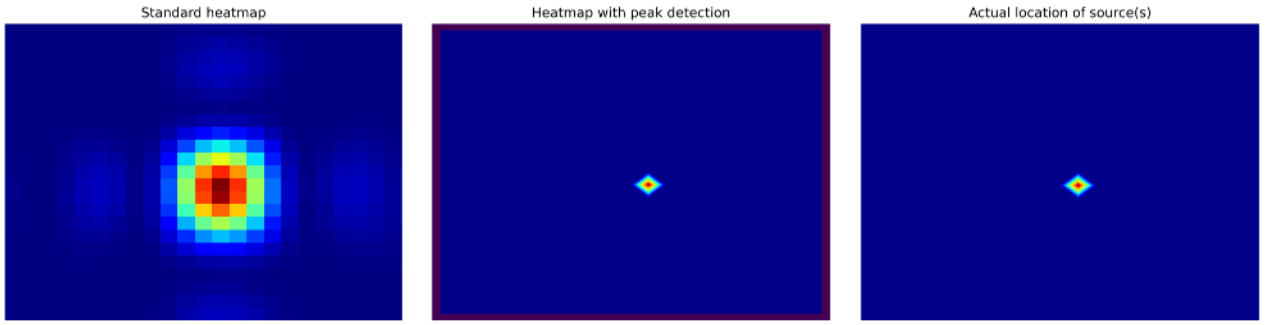
(a) (b) (c)

Figure 40: Simulation using one array and one source at: $f = 500$ Hz, $\theta = 20°$, $\varphi = 45°$.



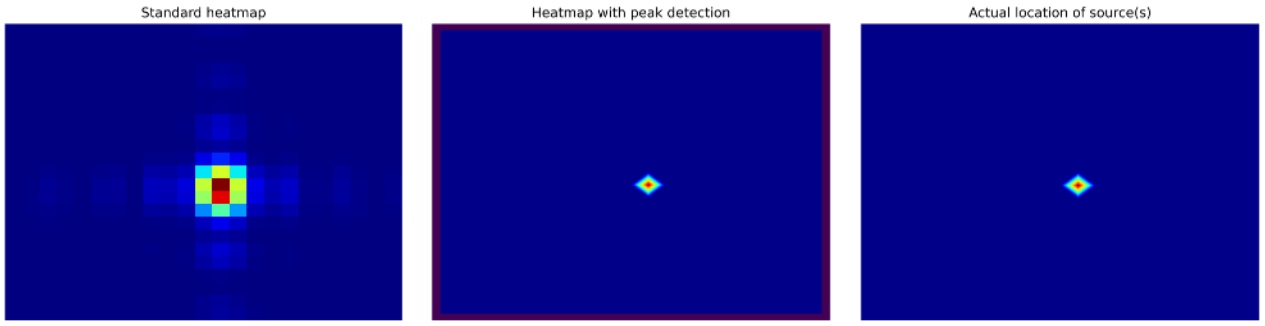
(a) (b) (c)

Figure 41: Simulation using one array and one source at: $f = 3000$ Hz, $\theta = 28°$, $\varphi = 0°$.



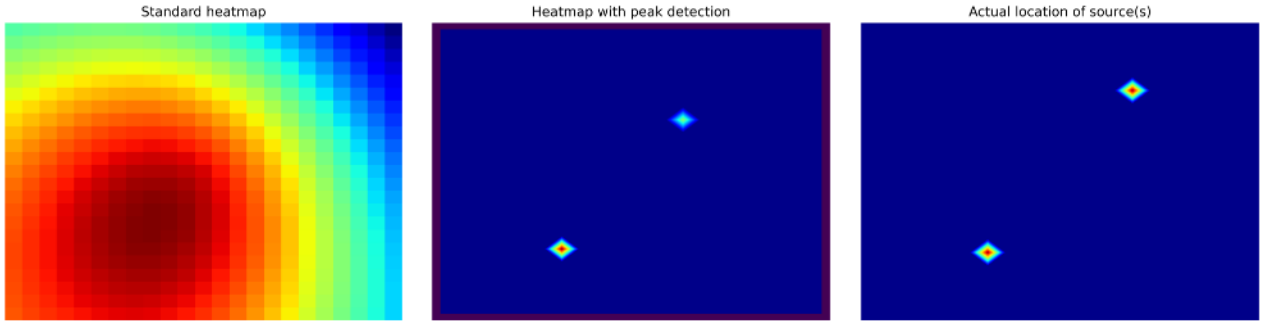
(a) (b) (c)

Figure 42: Simulation using one array and one source at: $f = 7500$ Hz, $\theta = 5°$, $\varphi = -45°$.



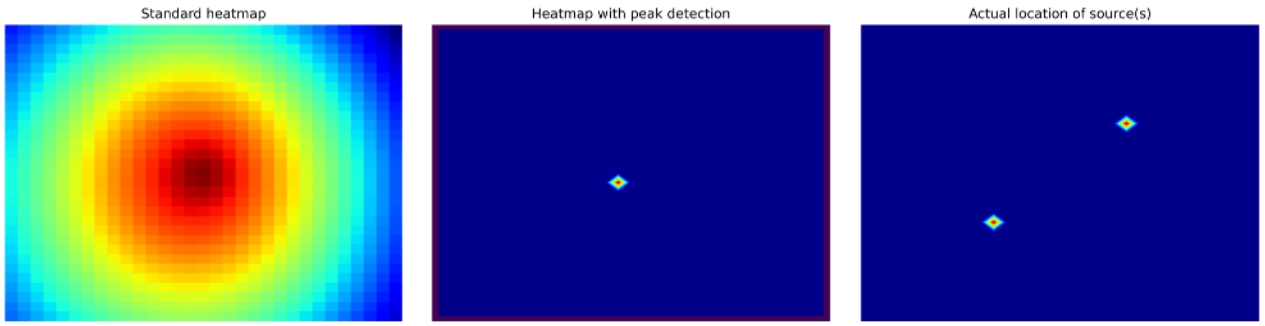
(a) (b) (c)

Figure 43: Simulation using one array and one source at: $f = 14000$ Hz, $\theta = 5°$, $\varphi = -45°$.



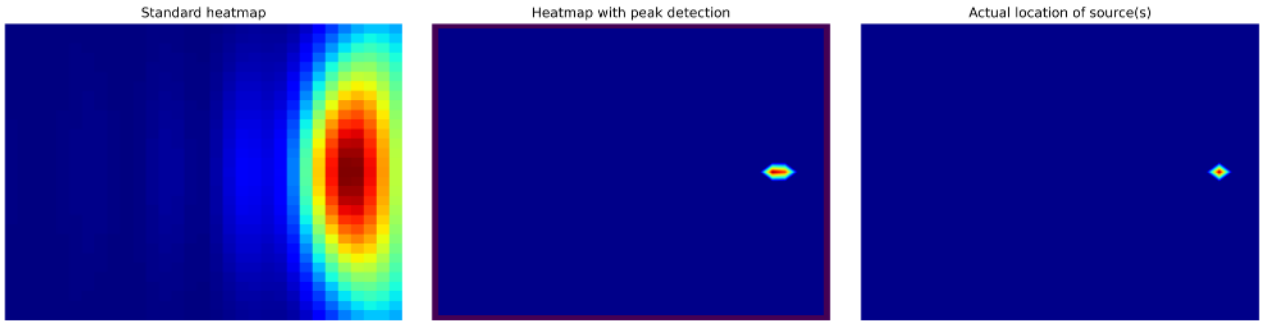
(a) (b) (c)

Figure 44: Simulation using one array and two sources at: $f_1 = 200$ Hz, $\theta_1 = 25°$, $\varphi_1 = 45°$ and $f_2 = 500$ Hz, $\theta_2 = -20°$, $\varphi_2 = 45°$.



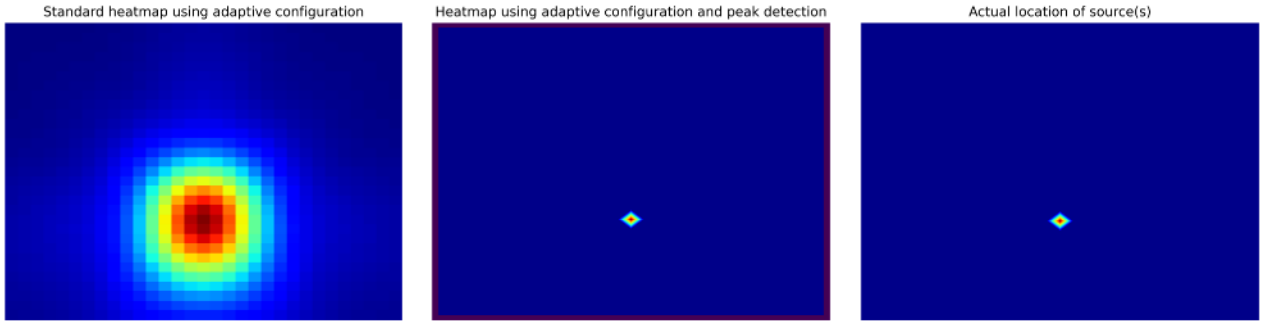
(a) (b) (c)

Figure 45: Simulation using one array and two sources at the same frequency: $f_1 = 300$ Hz, $\theta_1 = 15°$, $\varphi_1 = 35°$ and $f_2 = 300$ Hz, $\theta_2 = -15°$, $\varphi_2 = 35°$.



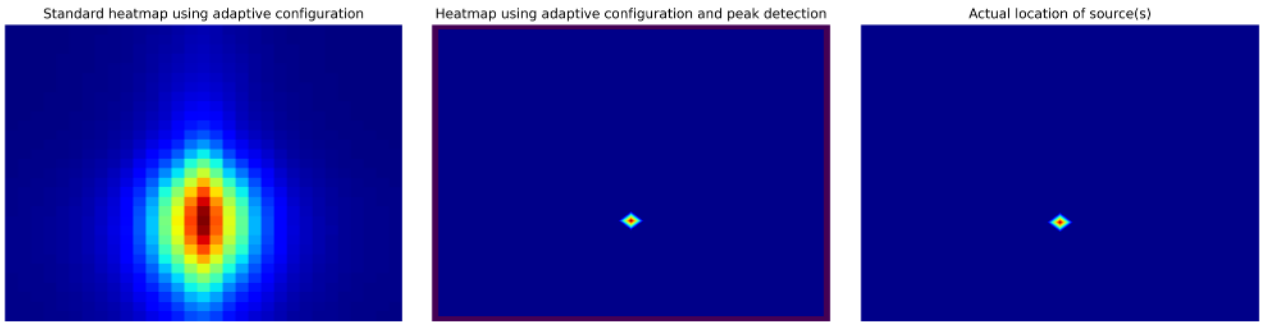
(a) (b) (c)

Figure 46: Simulation using three arrays and one source at: $f = 3000$ Hz, $\theta = 28°$, $\varphi = 0°$.



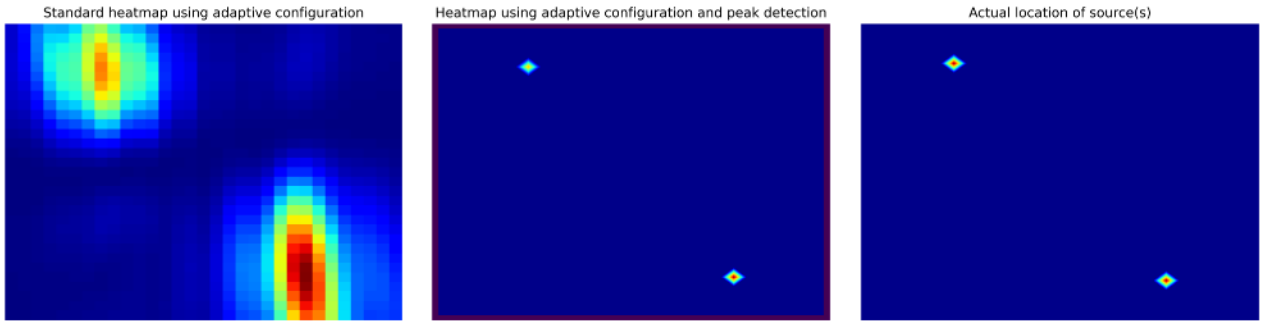
(a) (b) (c)

Figure 47: Simulation using adaptive configuration with one array and one source at: $f = [200, 8000]$ Hz, $\theta = 10°$, $\varphi = -90°$.



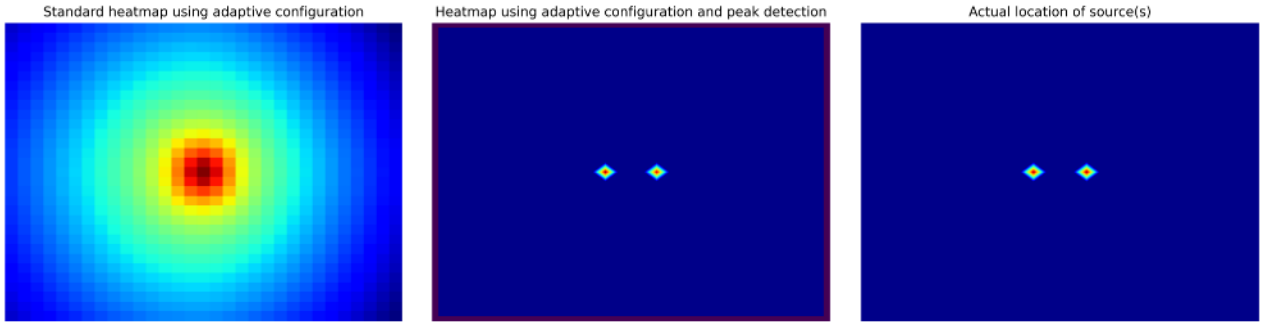
(a) (b) (c)

Figure 48: Simulation using adaptive configuration with three arrays and one source at: $f = [200, 8000]$ Hz, $\theta = 10°$, $\varphi = -90°$.



(a) (b) (c)

Figure 49: Simulation using adaptive configuration with three arrays and two sources at: $f_1 = [4000, 5000]$ Hz, $\theta_1 = 27°$, $\varphi_1 = -45°$ and $f_2 = [6000, 7000]$ Hz, $\theta_2 = 27°$, $\varphi_2 = 135°$.



(a) (b) (c)

Figure 50: Simulation using adaptive configuration with one array and two sources at: $f_1 = 200$ Hz, $\theta_1 = 5°$, $\varphi_1 = 0°$ and $f_2 = 300$ Hz, $\theta_2 = -5°$, $\varphi_2 = 0°$.