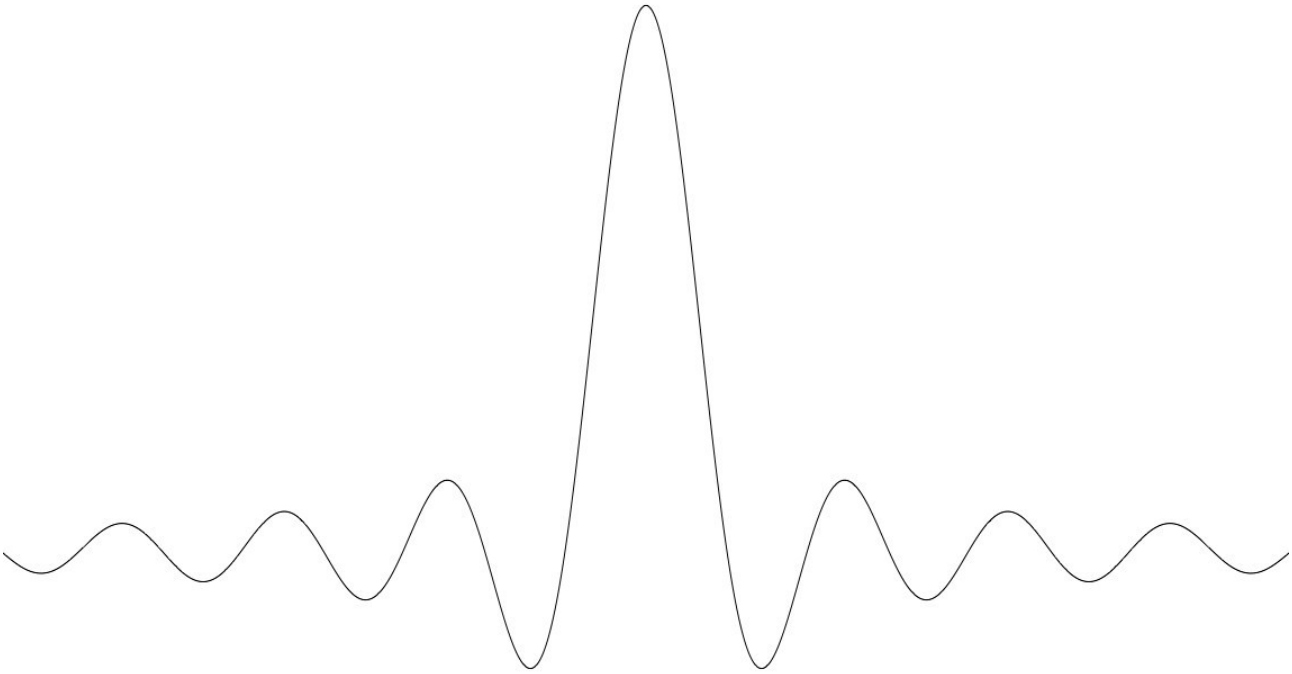


Microphone Arrays Domesticated:

Introduction of an easy to use, freeware beamforming VST plugin

Master thesis by Jan Stian Banas

AALBORG UNIVERSITY
DENMARK

Acknowledgement

I would like to express my gratitute to my supervisors, Francesco Grani and Stefania Serafin, who not only have been an everlasting inspiration, but most importantly patient guides on a journey through the unfamiliar depths of digital signal processing and spatial audio. Not only have they shown me established paths but also supported and encouraged to seek paths of my own.

I hereby would also like to thank teachers at Aalborg University in Copenhagen for creating a fantastic atmosphere for pursuing novel solutions and having always valued an open mind above all other virtues. Particularly I want to thank Cumhur Erkut for his dedication and expertise willingly shared with students; Dan Overholt for contagious enthusiasm, unbelievable practical knowledge and unmatched DIY spirit; Hendrik Purwins for endless patience and true desire to squeeze all skill reserves out of students.

I would not have reached this point if it weren't for you and I am grateful for quite an adventure that this education has been for me.

Contents

1. Introduction

In modern society, an evolution of technology has become a force responsible for stargazing of the youngsters, an urge to stay up-to-date of the adults and a lifeblood of many countries' economy. Developed by visionnaire researchers, novel inventions tend to have a crucial influence on strides that societies take on their path towards the future. While it is arguable, whether such progress is to be considered something explicitly gainful, there should be no doubt that innovation and science are what civilization as we know it is based upon.

In this contribution, development within media technology is considered. For the sake of following discussion, let us define media as all possible means of communication of information, including its generation, storage and broadcasting. Technology related to it has gone a long way since Gutenberg's printing press. In pursuit initially for spreading knowledge, various means of conveying messages have been developed, each following more revolutionary than the previous. A point has been reached where multimodal interfaces are becoming widespread, allowing users to engage in immersive virtual environments, that aim to mimic or even trascend the real world. Related technology is already easy accesible, but still not easy enough to use. Numerous researchers work hard on enabling access to virtual reality to the masses and at such high pace, it is a matter of very short time when homegrown applications will start emerging.

While it is common knowledge that in order to recreate an immersive experience multiple senses must be engaged, media technology is often associated (at least on consumer level) with visual cues. Ultra high resolution TV screens and head-mounted displays (or HMDs) get cheaper, and therefore easier accessible, significantly faster than groundbreaking inventions in the field of sound reproduction. Research in perception, however, shows that depending on the target experience, auditory stimuli play more important role than visual ones at times.

In the very foundation of this thesis lies a vision of the idea of sensor arrays, or more specifically microphone arrays, being something as popular as home recording is nowadays. Thankfully to rapidly advancing sound processing techniques, complex equipment that used to be unaffordable for most people can be easily simulated by a piece of software installed on one's laptop. Author of this thesis believes that with enough work towards simplifying the procedure and familiarizing the society with the concepts of spatial sound reproduction, home microphone array recordings will soon become ubiquitous.

2. Problem statement

This project serves a purpose of examining, whether conceptually advanced technology of microphone array recording can be popularized thankfully to introducing an user-friendly, open source piece of software for home use.

3. Analysis

In this chapter fundamental concepts related to array recording are explained. A thorough overview of current state of the art is presented together with some projects that are still in development.

3.1 Microphone Array

Multiple sensors operating within a single system, arranged in an imposed order is what a term sensors array represents. This concept has been long used in other fields, most notably to operate with electromagnetic waves (radar, radio astronomy, tomography) or ultrasound waves (sonar) [1].
In this contribution the author chooses to focus on microphone arrays, meant to work with acoustic signals. This choice is due to the fact, that for this purpose, array processing is something yet to be established, at least on consumer level (the author will further elaborate on this in State of the Art subchapter). However, while on superficial level it might seem like radio telescopes and microphones do not have much in common, the fundamental principles governing processing in both cases are similar, as they are related to wave propagation theory [2].

3.1.1 Wave Propagation

In order to have a good understanding of principles underlying sensor arrays, one must familiarize oneself with fundaments of waves physics. This subchapter serves a purpose of explaining those, with focus on concepts that are most relevant to the project.

3.1.1.1 Wave Equation

While the approaching sound as a perceived sensation is not to be neglected, physical aspects are more interesting for the scope of this contribution. As such, sound waves can be defined as propagation of a disturbance in air or other elastic media [3]. The wave motion is typically longitudinal in fluids. On a particle level, molecules move back and forth within some limits, creating regions of increased and decreased pressure. A single infinitesimal volume of the medium, assumed to have zero viscosity, behaves as represented in an equation:

$$\nabla^2 x(t,\vec{r}) - \frac{1}{c^2}\frac{\partial^2}{\partial t^2} x(t,\vec{r}) = 0 \qquad (1)$$

where vector $\vec{r}$ represents three-dimenstional coordinates of the volume and c is used to denote

the speed of sound, which for air is typically approximated to a value of $330\frac{m}{s}$. It is worth noting, that the above equation is widely used as governing various types of waves propagation, including electromagnetical waves and sound waves. In this article, the latter is of interest and consequently, $x(t,\vec{r})$ represents the varying sound pressure.

The solution to the above equation depends on the shape of the wavefront and is well known for fundamental cases such as monochromatic plane wave:

$$x(t,\vec{r}) = A\,e^{j(\omega t - \vec{k}r)} \tag{2}$$

or spherical one, originating from a point source:

$$x(t,\vec{r}) = \frac{-A}{4\pi r} e^{j(\omega t - \vec{k}r)} \tag{3}$$

In the above formulas A stands for the amplitude, ω is the radial representation of frequency and vector **k** is dependent on the wavelength as well as direction of propagation and is defined as:

$$\vec{k} = \frac{2\pi}{\lambda}[\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta] \tag{4}$$

θ and ϕ denote azimuth and elevation, respectively and λ is the wavelength.

3.1.1.2. Acoustic field

Although the above considerations apply to a monochromatic case (which is hardly present in real world), due to the linearity of given equation, they can be successfully extrapolated to a general case. According to the Fourier theory [4], any given signal can be decomposed into a sum of simple sines and cosines, as long as a so-called Fourier integral of it is convergent. This leads to a conclusion that any polychromatic signal satisfies the wave equation, derived for a single-frequency case.

Keeping in mind, that according to the abovementioned wave equation, sound is unambiguously described with merely two variables, that is time and space, it becomes clear that in order to reproduce a particular disturbance, one can either sample the signal temporally at a given location or spatially sample it at a given unit of time. While the former concept is common and easy to relate

to, as it has been a basis for sound reproduction every since the very beginning of electroacoustics (such as a monophonic loudspeaker in Thomas Edison's phonograph), the latter forms the basis for array processing techniques as it aims for capturing the signal at multiple locations during a single instant of time.

It is worth mentioning that this powerful concept not only applies to gathering information about wave propagation (or recording), but is also used for accurate sound field generation, using techniques such as wave field synthesis [5] or ambisonic [6].

3.1.2 Apertures

The term *aperture* is related to a area of space involved in receiving or transmitting a signal [7]. The most widespread application of this concept can be found in optical lens, present in every photo camera. Through control over the aperture diameter (typically approximately in the shape of a circle), the 'amount' of light let through to the sensor is limited, which results in control over the exposure to light. This phenomenon is mirrored in microphone array processing.

3.1.2.1. Definition. Directivity pattern.

In practice, each infinitesimal volume along the aperture is treated as a linear filter, with an impulse response of a(t,**r**). Consequently, the output signal is given by the convolution:

$$x_R(t,\vec{t}) = \int_{-\infty}^{\infty} x(\tau,\vec{r})\,a(t-\tau,\vec{r})\,d\tau \tag{5}$$

or alternatively in frequency domain:

$$X_R(f,\vec{r}) = X(f,\vec{r})\,A(f,\vec{r}) \tag{6}$$

In the above equation, A(f,**r**) is defined as the aperture function and is the fundamental parameter, describing the response in relation to spatial position on the aperture. This measure, however, does not take into account direction of arrival of the incoming waves. In order to include this, another parameter is commonly used, known as directivity pattern. In far field, it is basically obtained by taking Fourier transform of the aperture function:

$$D_R(f,\vec{\alpha}) = \int_{-\infty}^{\infty} A_R(f,\vec{r})\,e^{j2\pi\vec{\alpha}*\vec{r}}\,d\vec{r} \tag{7}$$

Direction of arrival is *hidden* in $\vec{\alpha}$ parameter, defined as:

$$\vec{\alpha} = \frac{1}{\lambda}[\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta] \tag{8}$$

It can be referred to as direction vector of the wave and has an important property of being dependent on the frequency.

In this project, a linear aperture is implemented and therefore it is useful to note the abovementioned formulas simplify to a singledimensional case. Furthermore, if it is assumed that a given aperture's proprietary function is frequency-independent, it may be written as a simple rectangular *window*:

$$A_R(x_a) \equiv \begin{cases} 1, |x| \le L/2 \\ 0, |x| > L/2 \end{cases} \tag{9}$$

where L stands for the aperture's effective length. For such a function, its Fourier transform, identical with its directivity pattern, is well known and has a solution given by:

$$D_R(f, \alpha_x) = Lsinc(\alpha_x L) \tag{10}$$

If the consideration is further simplified to horizontal plane only, a common formula for a polar plot is obtained:

$$D_N(f, \pi/2, \phi) = sinc(\frac{L}{\lambda}\cos(\phi)) \tag{11}$$

3.1.2.2 Discrete sensor array.

All the considerations from the previous subchapter serve a purpose of clarifying the mathematical foundations for microphone array. In practice, any sensor array can be considered as a discrete approximation of a continuous aperture and its properties are described by discretized versions of the abovementioned formulas. Hence for N sensors possessing each identical frequency response, spaced by d meters, the directivity pattern is equal to:

$$D(f, \alpha_x) = \sum_{n=-\frac{N-1}{2}}^{\frac{N-1}{2}} w_n(f) e^{j2\pi\alpha_x nd} \tag{12}$$

If only the horizontal plane is of concern, the equation transforms into:

$$D(f,\phi)=\sum_{n=-\frac{N-1}{2}}^{\frac{N-1}{2}} w_n(f)e^{j\frac{2\pi}{\lambda}nd\cos(\phi)} \tag{13}$$

It is clear from this formula that the array performance depends on three factors: the number of elements in the array *N*, the spacing between them *d* and frequency of interest $f=\frac{c}{\lambda}$. In analogy to a continuous aperture, it is observed that effective length of the array is given by *Nd* product. Theoretical analysis performed by plotting the directivity pattern for changing variable values leads to several important conclusions.

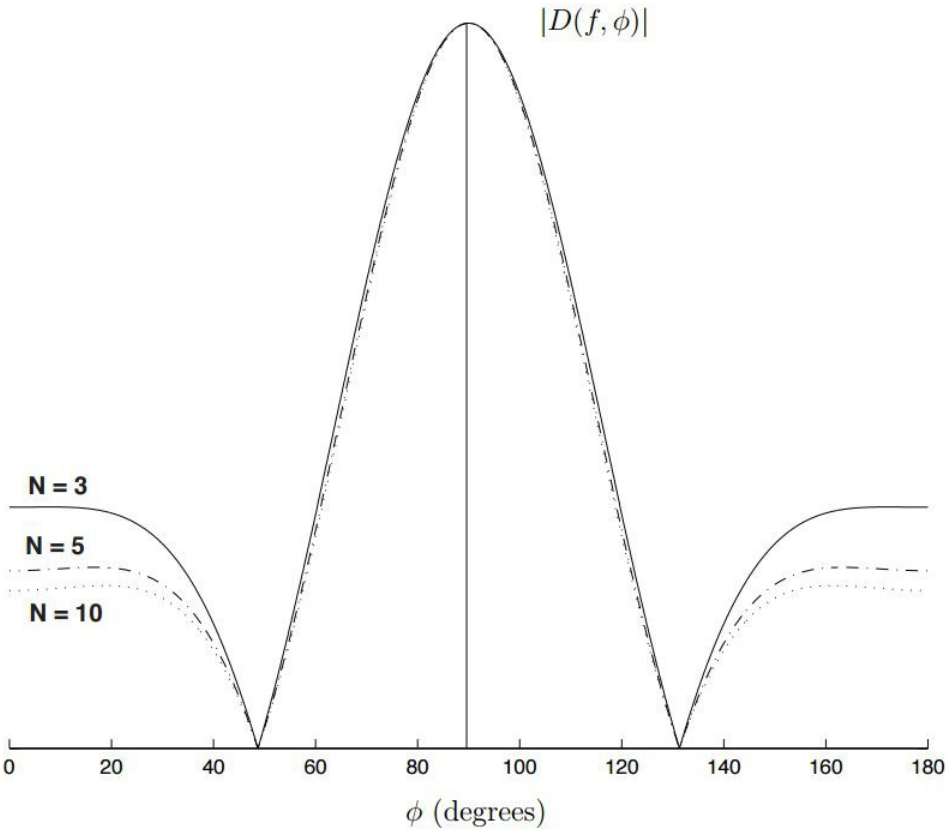


Figure 1. Varying N – the number of microphones in an array

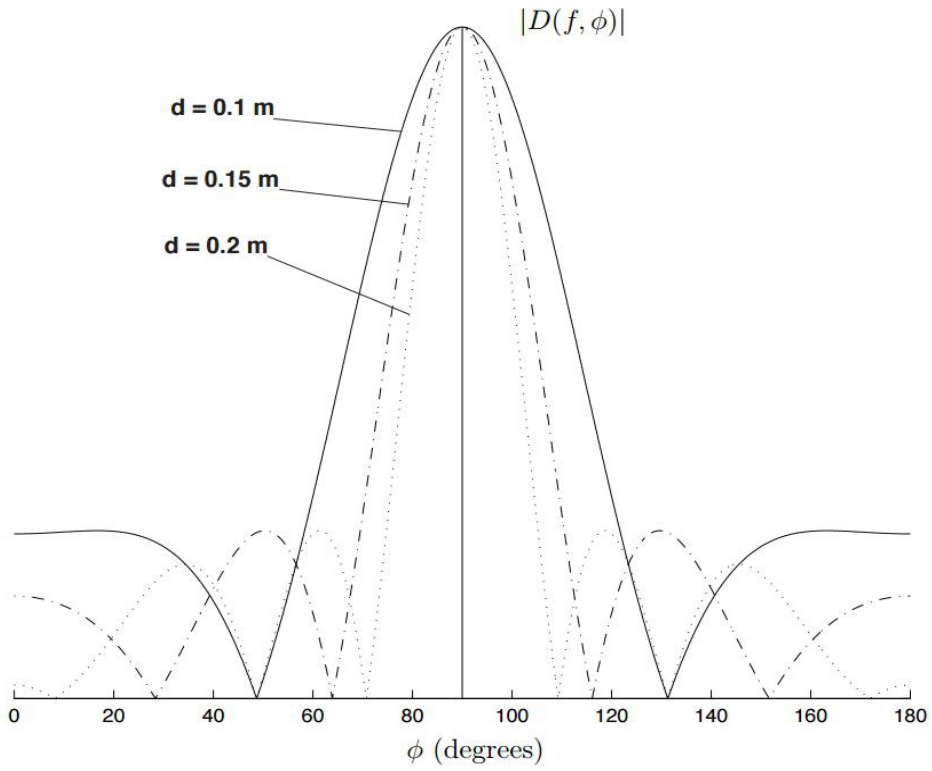


Figure 2. Varying d – spacing between array elements

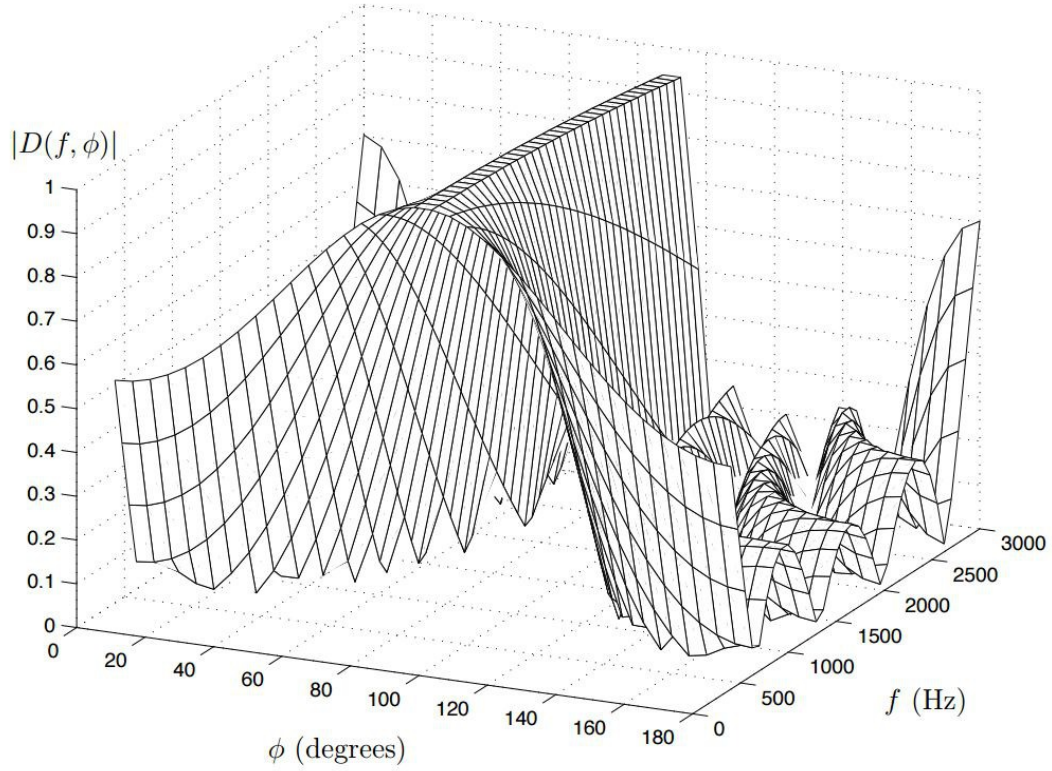


Figure 3. Directivity in function of frequency

First, it is observed that expanding the array in terms of the number of sensors has a positive influence on the sidelobe levels, meaning that the more microphones are there in the array, the more effective in supressing signals coming from undesired directions they become. Varying the distance between sensors results in widening or narrowing of the main lobe - the larger the spacing, the narrower the beam. In fact, the main lobe width is dependent on both spacing and frequency or to be more precised, it is inversely proportional to *fNd* product. Consequently, in order to obtain a constant beam width for all frequencies within a desired range, either spacing between or the number of microphones must vary along. One last observation from the above plots is that for increasing frequency, the main lobe width decreases significantly. These properties form a base for sub-array concept, described later in this chapter.

3.1.2.3 Spatial aliasing

When it comes to sampling, one must be aware of a phenomenon known as aliasing. As mentioned in 3.1.1.2 chapter, two types of sampling are commonly used - a temporal and spatial one. Practically, both of these apply to this project. While the concept of temporal sampling risks is well known and Nyquist theorem is something taught in the very first contact with digital signal representation, it might be not as obvious that the same danger occurs in spatial sampling and analoguous rules apply.

The temporal sampling theorem states that in order to preserve an unaliased frequency of value *f_max*, a signal must be sampled at a rate at least twice as high, that is:

$$f_s = \frac{1}{T_s} \geq 2\mathrm{f}_{max} \tag{14}$$

A similar requirement must be met when sampling a signal spatially. The difference is that sampling frequency is expressed in samples per meter, rather than samples per second and is equal to:

$$f_{xs} = \frac{1}{d} \geq 2\mathrm{f}_{xmax} \tag{15}$$

A given spatial resolution of a sensor array is considered sufficient if the following criterion is satisfied:

$$d < \frac{\lambda_{min}}{2} \tag{16}$$

This is a serious limitation and only very small spacing values allow to completely avoid spatial aliasing for the entire human hearing range.

3.1.2.4 Near-field sources

For the sake of brevity, it was assumed that sources were located in the far-field in previous considerations. The border between those two can be expressed with the following condition:

$$|r| > \frac{2L^2}{\lambda} \tag{17}$$

Wavefronts coming from above this value can be considered plane waves, while ones closer to the sensor are treated as spherical ones. This distiction is crucial when it comes to estimating a difference in distance travelled by the wave to arrive to two sensors. For far-field sources the wavefront shape is neglected, which, through simple geometry, leads to an assumption:

$$d' = ndcos(\phi) \tag{18}$$

where n is the sensor index in reference to zeroth one. Taking into consideration a spherical shape of the wavefront, the distance problem is approached differently, that is calculated as a difference between two distances, each drawn separately from the sound source. Assuming equally spaced microphones, for each pair of sensors, the relative distance travelled by a wave can be obtained by substuting *n* and *m* in the below equation with an appropriate index number:

$$d' = \sqrt{r^2 + 2rnd\cos(\phi) + (nd)^2} - \sqrt{r^2 + 2rmd\cos(\phi) + (md)^2} \tag{19}$$

The obtained result forms a base for calculating the desired time delay between sensors and the ratio of reference sensor distance to the n-th one serves as an amplitude attenuation parameter. The difference between far-field and near-field approaches is illustrated in Figure 4.

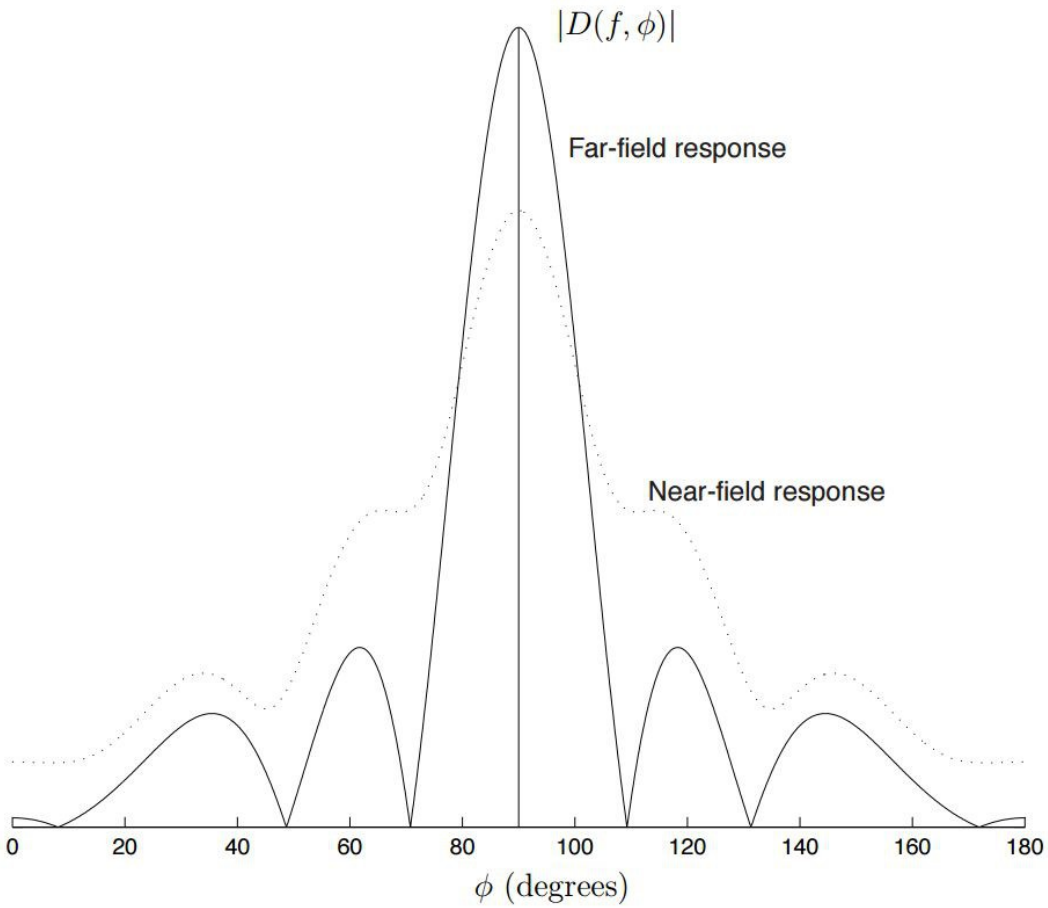


Figure 4. Significance of difference between far- and near-field approaches

3.1.3 Methods of array evaluation

In order to be able to compare various beamforming methods, common means for performance evaluation must be introduced [8]. In this chapter the most relevant ones will be described.

3.1.3.1 Array gain

This simple measure aims to emphasize a change introduced to the signal-to-noise ratio (SNR) between a single microphone and the array. Hence it is given by:

$$G = \frac{SNR_{array}}{SNR_{microphone}} \tag{20}$$

With an assumption of signal stationarity, the SNR is computed using the power spectral densities of signal and the average noise. For a single sensor, it is given by simple ratio between those two quantities. For the array output however, it is necessary to include the weighting of sensors as well

as their number in the calculations. For a homogeneous noise field, the resulting array gain is given by:

$$G = \frac{|\boldsymbol{W}^{\boldsymbol{H}}\boldsymbol{d}|^2}{\boldsymbol{W}^{\boldsymbol{H}}\boldsymbol{\Gamma}_{\boldsymbol{VV}}\boldsymbol{W}} \tag{21}$$

where **W** denotes a NxN matrix of frequency-domain filter coefficients proprietaty to the array, **d** is a N vector representing attenuations and time-delays at each sensor, $()^H$ stands for the conjugate transpose of a matrix and $\boldsymbol{\Gamma}_{\boldsymbol{VV}}$ is called the homogenous noise field coherence matrix and is given by:

$$\boldsymbol{\Gamma}_{\boldsymbol{VV}} = \begin{pmatrix} 1 & \Gamma_{V_0V_1} & \Gamma_{V_0V_2} & \cdots & \Gamma_{V_0V_{N-1}} \\ \Gamma_{V_1V_0} & 1 & \Gamma_{V_1V_2} & \cdots & \Gamma_{V_1V_{N-1}} \\ \vdots & \vdots & \ddots & & \vdots \\ \Gamma_{V_{N-1}V_0} & \Gamma_{V_{N-1}V_1} & \Gamma_{V_{N-1}V_2} & \cdots & 1 \end{pmatrix} \tag{22}$$

with frequency-dependent entries defined as:

$$\Gamma_{V_nV_m}(e^{j\omega}) = \frac{\Phi_{V_nV_m}(e^{j\omega})}{\sqrt{\Phi_{V_nV_n}(e^{j\omega})\,\Phi_{V_mV_m}(e^{j\omega})}} \tag{23}$$

It is worth noting that although the above formula (21) applies to a homogenous noise field, other assumptions can also be made by simply inserting a different coherence matrix to the equation.

3.1.3.2 White Noise Gain

This measure aims to evaluate the array's ability to supress an uncorrelated noise, which means noise with a correlation matrix equal to identity matrix. In this case, formula (21) becomes:

$$WNG(e^{j\omega}) = \frac{|\boldsymbol{W}^{\boldsymbol{H}}\boldsymbol{d}|^2}{\boldsymbol{W}^{\boldsymbol{H}}\boldsymbol{W}} \tag{24}$$

This parameter is particularly important when evaluating the array's performance in subject to self-noise of the sensors.

3.1.3.3 Beampattern

Beampattern is a measure that aims to evaluate the array response to a wave of specific frequency, coming from a specific angle (given by azimuth and elevation parameters). It is typically displayed on a logarithimc scale and can be easily calculated using the (21) equation, taking a logarithm with base 10 of it and multiplying by (-10) to obtain a decibel value. In such a case, the coherence matrix of a noise field is replaced by coherence matrix of a single wavefront with a given frequency ω and angle of arrival ϕ , θ , whose entries are calculated with:

$$\Gamma_{V_n V_m} = e^{j\omega\tau_{nm}} \tag{25}$$

additionally utilizing sampling frequency *fs*, with a parameter τ_{nm} given by:

$$\tau_{nm} = \frac{f_s}{c}\left(d_{x,nm}\sin(\theta)\cos(\phi) + d_{y,nm}\sin(\theta)\sin(\phi) + d_{z,nm}\cos(\theta)\right) \tag{26}$$

While it is impossible to plot full beampattern in two dimensions, as it depends on three variables, line arrays have a property of rotational symmetry and are therefore independent of ϕ . A sample sample beampattern is shown in Figure 5.

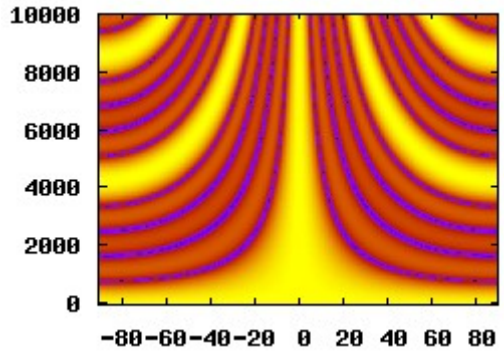


Figure 5. A sample beampattern (source: http://www.labbookpages.co.uk/audio/beamforming/delaySum.html)

3.2 Beamforming

In this chapter common techniques for beamforming are introduced and compared. The known methods can be categorized in fixed and adaptive classes, former of which require the direction of arrival to be known while latter possessing a property of being able to adapt to the varying input.

3.2.1 Fixed beamforming

3.2.1.1 Delay-and-sum

The most conceptually straightforward approach to beamforming relies directly and solely on the fact, that it takes a different amount of time for the wave to travel to utilized sensors. If the main lobe angle is to be rotated by ϕ radians, then delay applied to n-th sensor should be equal to:

$$\tau_n = \frac{\sqrt{r^2 + 2 r n d \cos(\phi) + (nd)^2}}{c} \tag{27}$$

In the next step, the delayed channels are summed to form an array output. In order to preserve the unity gain, each addend is multiplied by 1/*N*. The final formula in time domain is therefore:

$$y(t) = \frac{1}{N} \sum_{n=1}^{N} x_n(t - \tau_n) \tag{28}$$

or equivalently in frequency domain:

$$y(f) = \frac{1}{N} \sum_{n=1}^{N} x_n(f) e^{-j 2\pi f \tau_n} \tag{29}$$

Delay-and-sum beamformers are easy to implement and do not require powerful processors to work well. However, from performance perspective, they have proven to be underwhelming, with relatively wide main lobe, high sidelobe level and poor ability to suppress low frequency noise from any direction.

3.2.1.2 Sub-arrays

One way to improve delay-and-sum beamformer's performance is to implement it in a form of so-called sub-array. Rather than having a given number of sensors spaced equally, this concept relies on variable spacing and utilizing specific groups of microphones for limited bandwidths. As shown in chapter 3.1.2.2, spacing plays a significant role in shaping the directivity of a beamformer. One way to implement such idea is to form an array in logarithmic spacing [9]. Moreover, microphones should have filters applied, so that they only are active for a desired wavelength. The design process for such a beamformer is explained in detail in given reference and resulting system is proven to be

able to preserve a constand main lobe width for a wide frequency range, even for relatively small number of sensors involved in each band processing. However, even with such approach other limitations arise - the array itself has large physical dimensions (nearly 5m wide in order to be able to process frequencies down to 300Hz) and is built of multiple microphones, making the whole installation very unhandy and costly. Hence a need arises for more effective methods, such as superdirective beamformers.

3.2.1.3 Superdirective beamformers

It was shown in previous reasoning, that directivity, seen as degree of noise attenuation, is directly proportional to *N*, the number of sensors. In theory, if the spacing between microphones approaches 0 and the noise field is diffuse, this relation can be improved to be proportional to N^2. The so-called superdirective beamformers aim to take advantage of this property. Design of a particular beamformer depends on a given parameter to be optimized (please refer to chapter 3.1.3) and also of an assumed noise field [8]. Typically, in superdirective beamformer design the desired weighting coefficients of a transfer filter matrix is obtained by maximizing the array gain, that is factor of directivity [10]. The solution to this problem must be considered under at least two constraints: ensuring unity gain (so that the processing does not result in gain increase) and minimizing white noise gain (so that self-noise of microphones is not amplified). The resulting coefficients matrix is called the Minimum Variance Distortionless Response (MVDR) beamformer and is given by:

$$\boldsymbol{W}_c = \frac{(\boldsymbol{\Gamma}_{VV} + \mu \boldsymbol{I})^{-1} \boldsymbol{d}}{\boldsymbol{d}^{\boldsymbol{H}} (\boldsymbol{\Gamma}_{VV} + \mu \boldsymbol{I})^{-1} \boldsymbol{d}} \tag{30}$$

where μ is a scalar representing the ratio of sensor noise to the ambient noise power, $\boldsymbol{\Gamma}_{VV}$ stands for the coherence matrix as defined in (22) and **d** is a desired signal model, given by:

$$\boldsymbol{d}^{\boldsymbol{T}} = [a_0 e^{-j\omega\tau_0}, a_1 e^{-j\omega\tau_1}, \cdots, a_{N-1} e^{-j\omega\tau_{N-1}}] \tag{31}$$

where a_n denotes attenuattion on a sensor, calculated as a ratio between distance from the source to n-th sensor and to the reference sensor ($a_0 = 1$) and τ_n was defined in (27).

The above design applies for fixed scenario, when the number of microphones is known as so is spacing between them, assumed is identical specification as well as the type of noise field. What can be seen as a bridge between this approach and an adaptive one is a particular implementation structure, which utilizes a decomposition idea. The processing is done in two paths, one performed

on entire time-aligned input and the other on noise-only containing subspace. In order to achieve the desired result, a blocking matrix has to be implemented, fulfilling the criteria:

- size of (N-1)xN
- sum of all values in a row equals 0
- rank of N-1

This idea was first proposed by Griffiths and Jim [11] with the original blocking matrix design:

$$\boldsymbol{B}=\begin{pmatrix} 1 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 1 & -1 \end{pmatrix} \tag{32}$$

3.2.2 Adaptive beamforming

3.2.2.1 Generalised Sidelobe Canceler

The implementation method for beamformers described in previous chapter was described as belonging both to fixed and adaptive families. This is a subject to the nature of filter applied to the noise-only subspace of the input signal. In case, when it is an adaptive least-mean square filter [12], fed back with the array output, it can be considered a successful implementation of an adaptive beamformer (please refer to Figure 6). As a result, GSC is works both as a fixed and adaptive system, thankfully to two separated processing paths. The output of this type of a beamformer is obtained through subtraction of both paths.

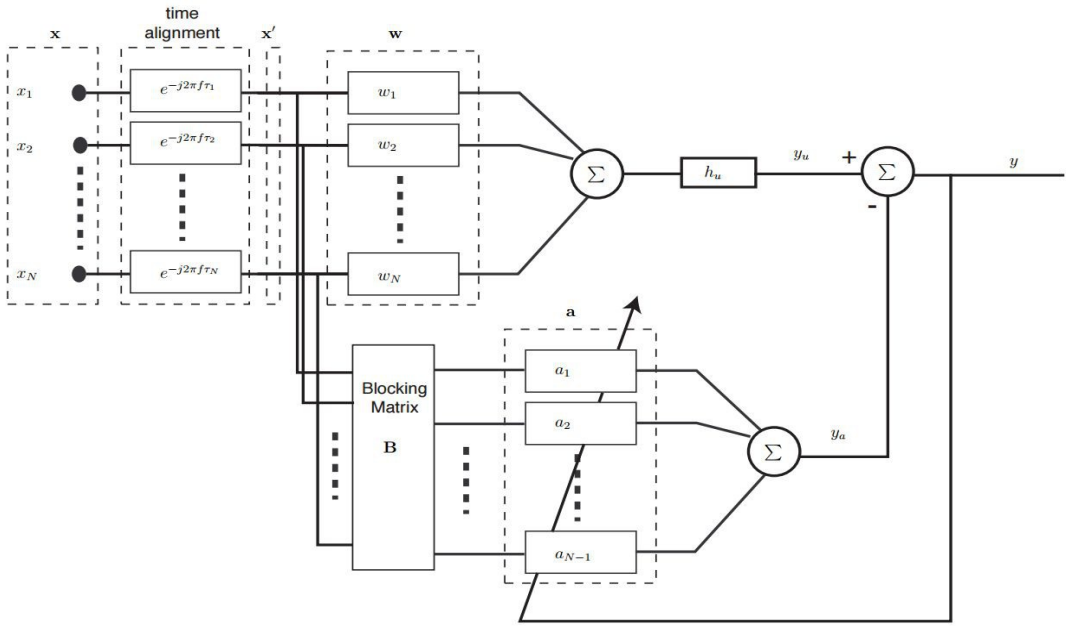


Figure 6. A Generalized Sidelobe Canceller schematic

The described system belongs to a family known as Linearly Constrained Minimum Variance (LCMV) beamformers. This is an approach more refined than simple least-mean square algorithms, as it takes into account a constraint of maintaining a specified transfer function for the desired signal. Thankfully to this, it is guaranteed that the response of the system has constant gain and linear phase.

While being the most common implementation of a beamformer, GSC is not flawless. The most significant danger is associated with a phenomenon called signal leakage, which occurs when the blocking matrix fails to separate noise from the desired signal. The broader the range of interest, the harder task it becomes.

3.2.2.2 Post-filtering

Although the abovementioned techniques provide an optimal theoretical solution for an assumed noise field and a narrowband desired input signal, they can be further improved for a broader range of applications with post-filtering [13]. It should be clear, that this approach is not suggested to be used *instead* of the MVDR beamformer, rather *in addition* to it. It is desirable to exploit knowledge gained about the signal to its full extent, that is not only for spatial, but also for frequency filtering. The improvement associated with this technique exhibits most significantly by cancelling any incoherent noise and enhancing spatial filtering by further suppressing even correlated noise sources. It also diminishes the array's performance vulnerability to minor errors in pointing direction. These properties, however, are strongly linked to efficiency of the beamformer itself and

hence do not apply if the former processing's performance is underwhelming.

3.3 State of the art

Now that theory behind microphone arrays is clarified, the author will peer into the current state of the art implementations. While it takes both hardware and software to take full advantage out of array processing, in this chapter they are discussed separately, as not always designers develop a full solution. It is more common for the hardware producer to deliver matching software as well than for software developers to deliver appropriate pieces of hardware. All below descriptions are based on insight into technology provided by producers.

3.3.1 Hardware

3.3.1.1 TSL Soundfield microphones

This manufacturer specializes in spatial audio solutions and delivers a full range of products, targeted mainly at film and TV industry sound engineering [14]. With this approach associated are some limitations and consequently even the most advanced of their products offers merely a supposedly 7-capsule array, achieving accurate sensing from an omnidirectional point W as well as X, Y and Z dimensions (it is a standard known as first order Ambisonic B-format). Matching processors offer rendering to 5.1 and 7.1 surround sound as well as stereo or monophonic downmixing. The flagship product is called SoundField DSF-2 Microphone System and consists of both a multichannel microphone and a processor, offering features such as variable polar pattern and the abovementioned variable format rendering. The price tag is ca. 19500 USD (17400 EUR). On the other hand, the cheapest piece of equipment offered by TSL is called the Soundfield SPS200 Software Controlled Microphone and is a tetrahedral transducer, exploiting a reduced-capsule B-format implementation [15]. It does not come with a matching processor, but is shipped with a VST plugin developed specifically to run with it on any DAW, which makes it more portable and compatible. The price tag is ca. 3000 USD (2700 EUR).

3.3.1.2 ClearOne Beamforming Mic Array

This producer offers a specialized microphone array, built with 24 capsules mounted on a flat surface [16]. This equipment is dedicated for mounting in conference rooms and such and is advertised as having an adaptive beam steering incorporated. It means that with such solution, very clear audio pickup is ensured in limited space, without a need for external mixing. The output is

either in mono or stereo format. With price tag of ca. 3000 USD (2700 EUR) it shows some market potential, provided it is an ultimate solution for conferencing. However, this impression is ruined by the fact that in order for this array to operate, consumer must invest in a dedicated processor as well, priced at around 9000 USD (8000 EUR).

3.3.1.3 Polycom Ceiling Microphone Array

The Polycom brand owner offers solutions for offices that need to be suited for conferencing. Among their line of products a spherical microphone array can be found [17]. Inside a single sphere, there are three cardoid capsules, calibrated to cover 360 degrees around their location. The microphone is advertised as being able to pick up sounds from an area of roughly 37 square meters aroud it (400 square feet). However, apart from being omnidirectional, this system does not offer any interesting features, unlike the abovementioned products. It might be considered a good value equipment though, as it can be bought for ca. 800 USD (700 EUR).

3.3.1.4 Core Sound Tetra Mic

It is an interesting product, embodying the tetrahedral microphone implementation for ambisonic recording at an affordable price [18]. It also has a custom piece of software designed in form of a VST plugin to enable all the desired post-processing. Moreover, as reported at the website, this microphone can be coupled with an iPad for portable ambisonic recording. It is not entirely flawless, as it offers rather underwhelming frequency response (40Hz-18.5kHz for 4dB tolerance, as reported by the manufacturer) and significant self-noise (19 dBA per capsule), but as stated before, these compromises serve a purpose of making this gear affordable. And consequently, the microphone itself is offered for 1000 USD (900 EUR), and shipped with all the required pieces (such as phantom power adapter, wires and a mount) is priced at ca. 1300 USD (1150 EUR).

3.3.1.5 mh acoustics Eigenmike

A very impressive, 32 high quality capsule loaded spherical array with not less inspiring capabilities [19]. According to the manufacturer, this product exploits a novel approach to beamforming, called Eigenbeams. These are component beams, consisting of a given subset of signals, coming from given microphones. These portions are later summed to obtain a desired directivity, also enabling focus at multiple sources. This approach allows full flexibility, as the user may choose to either take advantage of steering of the beams in real time, while recording; or the entire sound field might be captured as 32 separate audio streams and later processed by a dedicated piece of software for

optimal results. This is a well established product, well known for its high quality. However, although there are sound samples posted on the website, they do not exhibit the array's ability to suppress sounds coming from unwanted directions. The system can be purchased for 20000 USD (ca. 18000 EUR) and is shipped as a full package, including a MacBook Pro to run the software.

3.3.1.6 Shure Microflex Advance

A new line of products by the popular microphone brand Shure [20]. It aims the same target as the beforementioned ClearOne products - an office space equipped for conferencing. The flagship product of the line is a ceiling-mounted square surface array, designed to blend well in office interior. The manufacturer has predesigned usability scenarios for this array, advertising a capability of handling up to 21 speakers (on up to 8 channels) with one device. A neat feature is incorporation of DANTE standard for connecting the device, allowing use of standard CAT 5E (Ethernet) cables to form expansive microphone array networks for large conference venues. Furthermore, the coupled software is designed to be browser-based, which certainly improves the system's ease of use, making it all-platform. The exhibited frequency response is declared to be 180Hz-17kHz which seems suitable for speech transmission. Self-noise of the array is reported to be equal to 11dBA and the available single beam width can be switched between 35 to 55 degrees. Other than the ceiling-mounted MXA910, the producer introduced also a tabletop array under a name MXA310, showing roughly similar capabilities in terms of spatial filtering but for smaller applications - requiring up to 4 independent transmission channels. The dedicated software is, as mentioned before, browser-based and called IntelliMix, but unfortunately not much more is known about it at the time of writing these words. The product itself is yet to be launched, according to the producer in summer 2016, hence the price tag also remains unknown.

3.3.1.7 Dev-Audio Microcone

This is a tabletop microphone array, developed as a start up company by Iain McCowan [21], author of the comprehensive tutorial, that formed a base for the previous chapters [22]. Not much is known about this product currently, as the project has been suspended due to the company being acquired by another one. It incorporated six microphones to form a compact, omnidirectional array for desktop use. It used to be shipped with dedicated, very user-friendly software and even a software development kit (SDK) for users to fiddle with the gear's capabilities (which is an approach that this article's author dearly relates to). Designed to record small group meetings and priced at ca. 360 AUD (230 EUR) it had every reason to become a success. Unfortunately, currently the company undergoes a transition and as far as the author can tell, there is no way to purchase this gear at the

moment of writing these words.

3.3.1.8 Bruel & Kjaer PULSE Spherical Beamforming

The Bruel & Kjaer system designed for acoustic measurements not only implemets a beamformer, but also couples it with an array of cameras for accurate space imaging [23]. There is no standard build of this device as, according to the manufacturer, each microphone array is designed and built individually to suit the customer's needs, but the number of microphone capsules varies between 36 and 50. The algorithms implemented are a filter-and-sum beamformer and a B&K patented algorithm called SHARP. The frequency range is limited from 250Hz to 8kHz which is not an impressive score. By description, this device is designed for noise measurements, most notably inside vehicles, which might explain the poor frequency performance to some extent. The price tag for this system is not available as it is custom built.

3.3.1.9 Zylia Audioimmersion

This is a strongly inspired by EigenMike startup project [24]. Alike the predecessor, it embodies a spherical microphone array built with 32 capsules and is shipped with matching software. It is targeted for musicians, looking for a single tool to record rehearsals and shows with a possibility of instruments separation. The product is still in development, but the prototype demo shared on the company's website exhibits impressive performance when handling a single guitar and singing voice, located on the opposite side of the device. Further testing is in progress and more of them will surely demonstrate how well can this device perform. The price tag is not known yet but according to the manufacturer, it is aimed to be significantly cheaper than the competitor, EigenMike.

3.3.2 Software

In this subchapter only standalone software will be discussed - not coupled with a dedicated piece of hardware, but rather able to be used in miscellaneous scenarios, not requiring specific expensive gear.

3.3.2.1 Phased Array System Toolbox for MATLAB

A powerful and well equipped toolbox to work with MATLAB software, designed mostly for electromagnetic sensor arrays [24]. However, as it was discussed in previous chapters, the same

principles that govern electromagnetic waves can be successfully applied to acoustic waves. This piece of sofware however, like most MATLAB implementations in general, is designed for modelling of the arrays, rather than for real time processing. This makes it a thorough learning tool, allowing for easy creation of the virtual arrays and viewing their properties on polar plots as well as comparison between different types of beamformers. The toolbox is very intuitive to use and comes with a number of useful examples that help understanding ideas behind sensor arrays. The only downside, apart from the beforementioned unability to work in real time, is the price, as Mathworks ask 1350 USD (1200 EUR) for the full license. Fortunately, there is a 30 day trial version available.

3.3.2.2 VOCAL Acoustic Beamforming Software

VOCAL is a software company offering solutions for particular problems, designing their software to fully suit the customer's needs [25]. Therefore, not much can be said about their implementation of beamforming, as it is written on their website that a wide range of approaches is offered to suit the application best. Given how extensive and comprehensive their research resources are [26], one might trust in the company's high competence. An interesting example of their approach is offering a GSC implementation with extended adaptivity by introducing a blocking matrix, whose parameters are subject to input. What also makes this manufacturer stand out from the crowd is multiple platform support as their solutions can be either applied to common Texas Instruments and Analog Devices processors, OS X and Linux running computers as well as mobile devices. Price is dependent on specific order and therefore cannot be quoted here.

3.3.2.3 BeamformIt - robust acoustic beamformer

This is an open source implementation of the delay-and-sum beamforming technique [27]. Unfortunately it is not supported by the author, Xavier Anguera [28] anymore and the available source code can hardly be built without a tutorial, at least on a Windows machine. There are some external dependancies missing and hence multiple errors occur. This is a pity as plain analysis of the provided source code shows some promise that this could have been an interesting tool for people interested in array processing to fiddle with the most fundamental technique with little money investment.

3.3.2.4 AURORA Microphone Array B-format converter

As the name suggests it, this software is created to work with any number of microphones to output an Ambisonic B-format [29]. The idea behind it is to obtain a full B-format without the need of

purchasing an expensive dedicated microphone - only down to one omnidirectional microphone is required as an input device for the plugin to simulate the full spatial impulse response with several measurements at various points of space. It is worth noting that the author responsible for AURORA software is Angelo Farina, a praiseworthy researcher and contributor in the field of applied acoustics and spatial sound, notable also for sharing his research results in an open manner. The collection of plugins listed on AURORA website is truly impressive as they are efficient, highlt specialized tools, available for free download [30]. The downside, at least for those that are not final versions, might be unintuitive usage of some of them, but with enough patience and documentation they offer a lot of fairly easy accessible functionalities.

3.3.2.5 mh acoustics EigenUnits and EigenStudio

mh acoustics deliver a complete solution for a spherical microphone array and together with EigenMike, introduced in the previous chapter, there is a multipurpose software shipped [31]. The manufacturer offers a solution that takes advantage of one of two possible approaches - the user can either use a standalone application that allows recording monitoring, as well as real time polar pattern shaping called the EigenStudio (illustrated in Figure 7); or it can be chosen to use EigenUnits - a modular bundle of plugins, each responsible for a single task related to array processing, including gain setting, three-dimensional beam steering and variable output encoding, among others. The software can be downloaded for free from the producer's website, but it requires the matching EigenMike for operation.

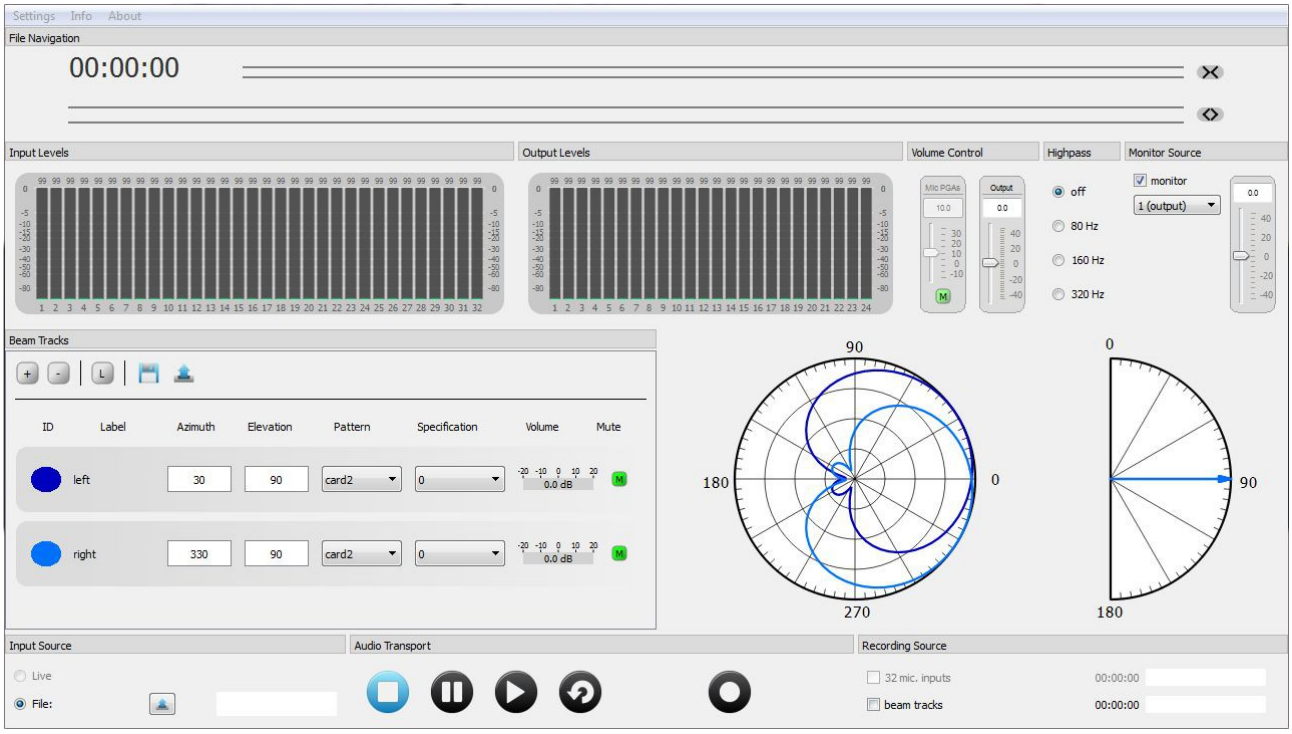


Figure 7. mh acoustics EigenStudio view

3.3.2.6 Dlab Array Beamformer

Another implementation of microphone array processing is offered by a Swiss-based company Dlab, which specializes in digital audio products [32]. Their ArrayBF is an algorithm designed to work with up to 4 microphones, spaced at 10-18mm from each other. It is advertised as enabling adaptive beamforming, with an exceptional feature of not only steering of the main lobe, but also the cancellation directions. The resulting output exhibits an improvement of up to 10 dB difference between main and side lobes levels. Another interesting feature is compensation for microphone mismatch, meaning that the user does not necessarily have to use identical capsules for their custom array. Depending on the processor capabilities, the user may choose between fixed or adaptive beamforming implementations. The price tag for this software is unknown.

3.3.2.7 STMicroelectronics osxAcousticBF

This is an extension for dedicated software used to program the STM32 microcontroller and to be coupled with a pair of digital MEMS microphones [33]. Equipped with an extensive documentation, this is likely a useful tool for the abovementioned development board owners who want to include beamforming functionality in their projects. According to information provided by the manufacturer, this implementation takes advantage of the adaptive beamforming algorithms to enable directivity steering of merely a two microphone array. The software can be downloaded and used for free, provided the user owns a dedicated piece of hardware.The owner doesn’t, but he appreciates the open availability of this particular and other tools designed for STM32 microcontroller.

3.3.2.8 DSP Soundware Acoustic Beamforming Software

Another multiplatform implementation of beamforming [34]. Information provided by the manufacturer is obscure and incomplete, as it is not clear what techniques are incorporated, other than fact that the software exhibits adaptive performance, with both beam and null region automatic steering. This solution is flexible in terms of number of microphones used and can be delivered to work on popular processors, as well as Linux/Windows machines and mobile devices (Android/iOS). When requested a pricing for this product, the author has received a response claiming that the project is under sustained development and therefore cannot be shipped at the moment.

3.3.2.9 Adaptive Digital Acoustic Beamformer

A solution designed specifically to work with speech signals and to be ran on three popular Texas Instruments digital signal processors [35]. It offers support for up to 8 microphones formining a line array with spacing not larger than 10 centimeters. Due to strictly foreseen applications, it only offers maximum sound frequency processing of 4kHz (as sampling rate is up to 8kHz). It can be chosen to use either fixed or adaptive steering mode.

3.4 Summary

From sensor array theoretical analysis it is clear that concepts related to this branch of science are well investigated and developed throughout years of research. Knowledge synthesized by many contributors has led to multiple attempts to implement beamforming algorithms and apply them to different types of microphone arrays. This branch of industry is still under development as results obtained so far often exhibit imperfect performance - whether it be in terms of frequency range, spatial resolution or down-to-earth costliness. Therefore a pursuit for new solutions continues and novel proposals arise, such as accounting for reverberation of a room [36], a concept known as differential microphone arrays [37][38] or an idea to generate virtual microphone channels by interpolating from existing ones [39]. The purpose of this project, however, is an attempt to make microphone array usage and design more available. So far, for a homegrown sound engineer interested in spatial filtering from a practical perspective, it is rather difficult to get started. The state of the art equipment and software tends to be costly and user-unfriendly. It is hardly possible to gain an empirical understanding of what can and cannot be achieved with such solutions without a serious time and interest investment. It is the author’s profound desire to change the way things are and verify the meaningfullness of such attempt.

4. Design

In this chapter, decisions concering design of the software being the substance of this contribution will be discussed and justified. This scope involves choice of final platform for the application as well as development environment, also the beamforming technique and particular parameters to be manipulated by the user.

4.1 VST Plugin

4.1.1 Introduction

Virtual Studio Technology (VST) is a standard developed by renowned pioneer audio software and hardware producer, Steinberg. Introduced first in 1996 together with native digital audio workstation (DAW) Cubase 3.02, it has gradually grown to become the most widespread format for third-party audio applications [40]. Among other features, it owes its popularity to a software development kit (SDK), that was released simultaneously. It encouraged developers and designers around the world to implement their ideas for audio software in a standarized way, so that the resulting application would be compatible with most hosts. The built VST plugin can be easily inserted into any DAW of choice, so users don't have to instal and learn how to use new environments every time they want to try out some exciting new synthesizers or effects. A subtype of VST are the so-called VSTi, where *i* stands for *instrument*. These can take MIDI messages as input and generate audio stream to the output of the system. The current version of SDK goes under a number 3.6.5 and enables 64-bit audio processing, multiple inputs and outpus (also for MIDI) as well as GUI dedicated classess, among others.

4.1.2 Compatibility

As mentioned before, major part of VST success is its flexibility, as there is no limitation to a single software they can be coupled with. However, although in theory these plugins can be ran on Windows, as well as Linux and OS X, in practice users often encounter difficulties when trying to host them on Apple machines. This is due to the fact that Apple supports their proprietary competetive format, known as Audio Unit (AU). VSTs can still be ran on OS X, but developers seldom take advantage of this property, prefering to build the native AU format instead. As for Linux systems, very few developers target their work to be compatible with those, mainly because of little commercial potential. The reasoning behind choosing to create a VST plugin rather than an AU one or a standalone application is common familiarity with the concept, covenience related to using one, favourite host (DAW) for all audio projects and finally the author himself being a Windows machine user, rather than an Apple. Keeping in mind that eventually it is desirable for a piece of software to be cross-platform, the author began looking for an environment that would enable this feature. As it turned out, one overall beneficial way to be able to easily develop audio software compatible with many platforms is to make use of JUCE framework, introduced in the following chapter.

4.2 JUCE

4.2.1 What is JUCE?

With the name being an acronym for Jules' Utility Class Extensions, JUCE is a very powerful toolkit with emphasis mostly on audio software development, which has been started by a single master coder, Julian 'Jules' Storer around 2003. It began as a package created to support the Traction DAW and later in 2004 separated from this project to form an independent entity, both on development and business level [41]. It is a complete tool suitable for many applications, from digital signal processing, through GUI elements to graphics applications. Multiple classes available as a part of the package enable the even relatively unexperienced designer to achieve their goals in a fairly easy way, without having to deal with low-level issues, like thread management or such. Moreover, for non-commercial usage, the package is available under GPL license, which makes lowers the investment required to get started with it only to time spent learning its features.

4.2.2 Features

There are many advantages of this framework, on various levels. The essential requirement for a piece of software chosen to work with is for it to be able to do what the designer wants in an intuitive way. For audio applications, JUCE possesses pretty much everything that one could ask for, including sound card input and output handling, filters and basic effects implementations, Open Sound Control (OSC) and MIDI support, reading from various audio file formats and many more (for a complete list of classes and related documentation, please refer to [42]). On top of that, the library comes with a dedicated project management tool, Projucer, which embodies a graphic interface for control over basic project parameters, such as type (a standalone application, a plugin, a library), the number of supported inputs and outputs, architecture (32 or 64 bit) and more. A crucial asset of using Projucer is also the ability to release application for different platforms, using the very same audio processing code. These include Windows, Linux, OS X, iOS and Android. The code itself can be modified inside Projucer or in the designer's IDE of choice (popular environments are supported, such as Visual Studio, Xcode, Android Studio and Code::Blocks). If one chooses to purchase a commercial license, a live coding feature is enabled, allowing usage as real-time programming performance tool (also supporting sound). In future versions, Projucer is also supposed to support building projects without use of the third-party IDEs. Not to be neglected is also very easy to use GUI editor, allowing the developer to create a good looking interface with a given set of templates without having to write a single line of code.

4.2.3 Learning and support

Possible a huge part of JUCE success and rapidly increasing popularity is a significant number of resources available, shared free of charge on official website. Following the tutorials prepared by people at ROLI (a company holding JUCE ownership), even an unexperienced developer can make their first audio applications in no time. Naturally, these guides merely scratch the surface of what can be achieved with this framework, but they are more than enough to get started and excited with it. For any specific issues encountered during the experience with JUCE, developers might also browse the well-developed forum. If it so happens, that the sought answer cannot be found, asking a public question is very advisable, as the community, including the master creator Jules, takes effort to help solve even rookie problems and shows true dedication when it comes to solving more challenging ones to a degree that it is pleasurable just to read the discussion. All in all, the availability, multitude of features including multiplatform and multipurpose support and tools for learning as well as the community around it are the factors that pushed the author to pick JUCE as a tool to implement his idea.

4.3 Beamforming

Before going into real-time JUCE implementation, the author decided to try various beamforming algorithms in MATLAB simulation-enabling sotware. This served a purpose of experiencing the capabilities of different approaches as well as verifying the implementation complexity of them from an unexperienced developer perspective. An important design decision was to limit the software to only be able to handle one-dimensional line array processing, rather than two- or three-dimensional square, circle or spherical ones. This serious limitation was dictated by trying to keep things simple for first iteration of the project, while maintaining an open possibility of extending the software capabilities in the future. Initially, a simple delay-and-sum beamformer was implemented, designed to work with prerecorded audio tracks (more details on those can be found in Evaluation chapter of this thesis). The code which was used to verify the efficiency of this algorithm is depicted in Figure 8.

```
clear all
close all

fs=44100; %sampling frequency of the recordings
d1=0.02; %microphone spacing in case 1
d2=0.005; %microphone spacing in case 2
source=1; %sound source distance from the first microphone in the array (in meters)
fi=pi/4; %beam angle
c=330; %speed of sound

mic1_1=audioread('C:\Users\JAN\Documents\MATLAB\arrayRecordings\NAV101 - Mic 1.wav'); %make sure the path is correct
mic2_1=audioread('C:\Users\JAN\Documents\MATLAB\arrayRecordings\NAV101 - Mic 2.wav');
mic3_1=audioread('C:\Users\JAN\Documents\MATLAB\arrayRecordings\NAV101 - Mic 3.wav');

delay_ff=d2*cos(fi)/c; %far-field delay in seconds estimate
delay_nf=(sqrt(source+2*d2*source*cos(fi)+(source*d2)^2))/c; % near-field delay in seconds estimate
delay_s=abs(round(delay_nf*fs)); %delay in samples

for i=1+delay_s:1:length(mic1_1)-delay_s
    buff(i)=(mic1_1(i-delay_s)+mic2_1(i)+mic3_1(i+delay_s))/3;
end

audiowrite('result_nav_101.wav',buff,fs);
```

Figure 8. MATLAB implementation of delay-and-sum beamformer

Experimenting with numerous recordings has led to some interesting observations. Expectedly, the algorithm performs best for peak cosine values, so the most significant filtering is observed in end-fire configuration, that is when the angle of direction is equal to either 0 or π. With sampling rate of 44100Hz and spacing between microphones as high as 2cm, a maximum inter-microphone delay of 3 samples is applied, which, depending on sources positions, gives somehow satisfying results. Further discussion is presented in Evaluation chapter.

However moderately successful the initial approach was, the author, encouraged by supposed superiority of superdirective arrays over the basic delay-and-sum ones, made an attempt to implement such an array processor prototype in MATLAB as well. Following the derivation found in [8] has led the author to the implementation shown in Figure 9.

```
%% Beamformer in assumed diffuse noise field

input_matrix=cat(3,mic1_1_freq_trunc, mic2_1_freq_trunc, mic3_1_freq_trunc); %combine inputs into a single matrix
single_snippet=input_matrix(1,:,:); %sample buffer containing the array input

entry12=sinc(2*pi*bins_hz*d1/c); %coherence between noise at 2 close microphones
entry13=sinc(2*pi*bins_hz*2*d1/c); %coherence between noise at 2 further microphones
coherence=repmat(1,[3 3 window_size]); %initialize coherence matrix of size as big as the number of frequency bins
coherence(1,2,:)=entry12; %fill the matrix
coherence(1,3,:)=entry13;
coherence(2,1,:)=entry12;
coherence(2,3,:)=entry12;
coherence(3,1,:)=entry13;
coherence(3,2,:)=entry12;

a0=1; %attenuation on the sensors
a1=source/(source+d1); %formula: norm(source_location-reference_sensor_location)/norm(source_location-nth_sensor_location)
a2=source/(source+2*d1);
tau0=0; %delay on the sensors (in seconds)
tau1=(source-(source+d1))/c; %formula: (norm(source_location-reference_sensor_location)-norm(source_location-nth_sensor_location))/c
tau2=(source-(source+2*d1))/c;

tau0_s=round(tau0*fs); %conversion to samples
tau1_s=round(tau1*fs);
tau2_s=round(tau2*fs);

desired_signal_far=[ones(1,window_size); exp(-1j*2*pi*bins_hz*d1*cos(fi)/c); exp(-1j*2*pi*bins_hz*2*d1*cos(fi)/c)]; %following the model
desired_signal_near=[a0*exp(-1j*2*pi*bins_hz*tau0_s);a1*exp(-1j*2*pi*bins_hz*tau1_s);a2*exp(-1j*2*pi*bins_hz*tau2_s)]; %following the model

for i=1:window_size %compute weights
    weights_far(i,:)=(inv(squeeze(coherence(:,:,i)))*desired_signal_far(:,i))/(desired_signal_far(:,i)'*inv(squeeze(coherence(:,:,i)))*desired_signal_far(:,i));
    weights_near(i,:)=(inv(squeeze(coherence(:,:,i)))*desired_signal_near(:,i))/(desired_signal_near(:,i)'*inv(squeeze(coherence(:,:,i)))*desired_signal_near(:,i));
end

%% Apply beamformer
for i=1:length(input_matrix)
    output_far(i,:,:)=weights_far'*squeeze(input_matrix(i,:,:));
end
```

Figure 9. MATLAB prototype of superdirective beamformer implementation

Unfortunately, this attempt was not finalized due to a conceptual infelicity encountered at the point, when the filtered signal was supposed to be synthesized back to time domain from its frequency representation. More precisely, the obtained matrix dimensions appeared to lack physical meaningfulness. During the time allocated for this stage of project development, the author did not manage to resolve this problem.

Consequently, after obtaining acceptable results using the delay-and-sum beamforming approach and failing to prototype the other methods, it was settled that for the actual real-time implementation the former would be picked.

4.4 User experience

4.4.1 Controllers

As it has been emphasized throughout the scope of this thesis, creating an easy to use and smooth operating from a user's perspective software was an as important design criterion as the processing itself. One of the ways to improve user experience is definitely to avoid information overload. Therefore user should only be granted control over most essential processing parameters and those should be chosen carefully. The primary parameter of interest is definitely the directivity, as this is the very core of why would someone want to use a microphone array, to begin with. For this purpose, the author finds slider-type controller to be suitable, allowing smooth transition between the desired states. Alternatively, for each parameter, user may decide to type in a desired value, rather than dragging the slider. Alike controllers have been chosen to manipulate two other parameters that are included. First of them is spacing between microphones as it is a definitive hardware parameter of the array and its influence on processing is crucial with delay time is directly proportional to this quantity. The second is the number of microphones incorporated as the author wants to enable potential users to verify the relation between this parameter and the array performance. Finally, as for any other audio plugin, there must be a 'Bypass' switch, responsible for turning the functionality on and off.

4.4.2 Feedback

An analysis of existing beamforming software has given some insight into what do developers usually decide to include as feedback. The most spectacular is definitely polar plot, providing information about both the direction of main lobe and level interval from side lobes. It can be found for example in EigenStudio, the standalone piece to work with mh acoustics' EigenMike. On the

other hand, however good looking it is, the author thinks that it lacks visualization of the array itself - with microphone dimensions and spacing between them clearly intelligible. Therefore it is believed that the representation illustrated in Figure 10 would be more suitable for amateurs wanting to build their own microphone arrays (with a vector symbolically indicating the main lobe orientation).

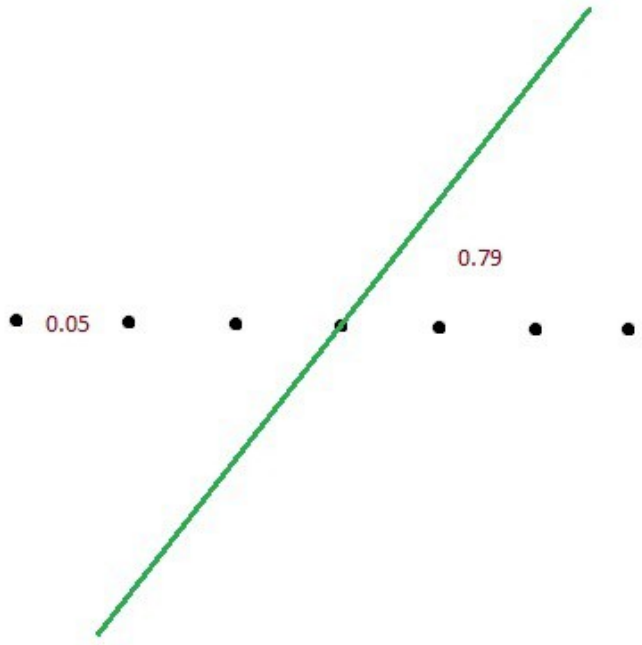


Figure 10. A prototype of visual feedback in the plugin

5. Implementation

Having decided upon creating a VST plugin using JUCE framework, knowing about the functionalities to be included and usability to be achieved, time has come to actually learn how to and implement the intended goals.

5.1 Getting started

With no prior C++ programming language experience whatsoever, it felt like a mandatory first step on a way to develop the VST plugin to learn the language fundamentals. Fortunately, there is plenty of helpful materials around the Internet so it turned out to be a straightforward and pleasurable experience to learn the basics. For that purpose, search for best courses has led the author to edX

online learning website, where introductory C++ course by Microsoft was hosted [43]. For more indepth information and further reference, the author often visited a written C++ tutorial, found on [44].

The next step on the road to audio plugin development was following the guidelines provided by JUCE in form of step-by-step tutorials, familiarizing apprentices with basic features of the framework [45]. From teaching how to use the Projucer project manager, through explaining the GUI functionalities to a comprehensive guide through audio processing dedicated functions. However, the introduction to creating plugins is limited to explaining how to manage MIDI related tasks. Therefore a need for further guidance arose, in order to increase confidence in plugin development.

As it turned out, there is a fantastic tutorial available, aimed precisely at non-MIDI VST plugin development using JUCE [46]. Made available by Redwood Audio, ‘an up and coming custom shop which designs and builds one of a kind instruments, audio effects and other audio-centric projects for the musicians who inspire them (and sometimes ourselves)’, which apart from their own designer and builder activity, also provide and gather useful resources for DIY enthusiasts to play with. These are divided in three categories, concerning wood craft (or luthiery), electronics and finally audio software development. The utilized tutorial is very well-written and intended for people having little experise in software development field. The project introduced in this tutorial formed a basis for this project’s beamforming implementation.

5.2 User interface

Using the built in Projucer GUI editor, the user interface was easily created. As recalled from the Design chapter, there were three parameter controllers to be implemented: direction of the beam, spacing between microphones and number of microphones in the array. For parameter value ranges, it was decided that angle should take values between 0 and pi with a 0.01 step, because of symmetry of directivity patterns [2]; spacing between microphones would be set to a value between 0 and 1 meter, with a step size of 1mm (0.001m) to maximize the educational value of the plugin, enabling verification of extreme parameter values’ influence on results; finally, possible number of input channels, corresponding to the number of sensors in the array can take values between 3 and 15 with a step of 2, so that only odd numbers are allowed (therefore simplifying calculations of delays with reference sensor always being the center one). The last implemented controller was the ‘Bypass’ switch, allowing to turn the processing on and off in an easy manner. Final version of graphical user interface can be seen in Figure 11.

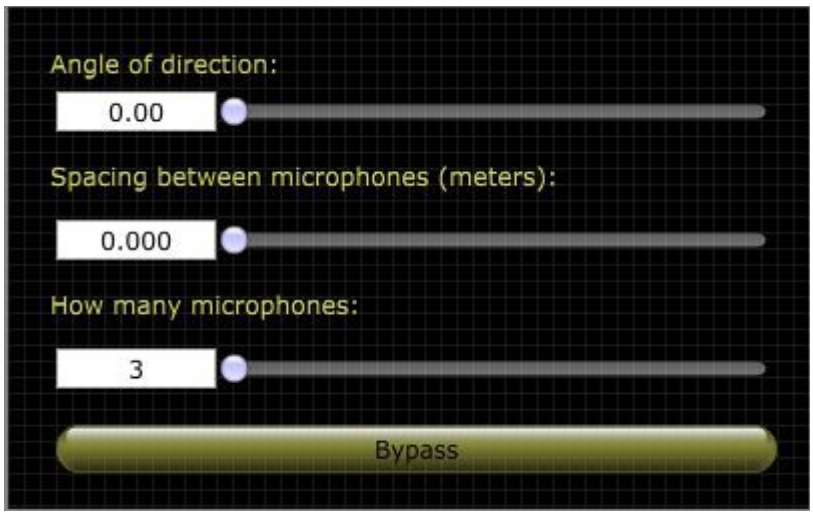


Figure 11. Graphical user interface of the plugin

Clearly there is no visual feedback implemented, even though it was designed to be there. This absence is mostly an effect of author having little proficiency in C++ and JUCE programming, however it can be argued that the simplistic look of current version has some charm. Therefore for next iteration of this project, it would be a good idea to implement visual feedback as mentioned in Design chapter, maybe in form of a context menu related to the angle parameter or simply a switchable sub-window.

As for the presentation, naturally there is C++ code that realizes this interface and it can be found in Appendix, but most of it is automatically generated by Projucer. The part that is not is the function called sliderValueChanged, responsible for sending values read from the slider to the processing part of the program and it is displayed in Figure 12.

```
void BeamformerAudioProcessorEditor::sliderValueChanged (Slider* sliderThatWasMoved)
{
    //[UserslidervalueChanged_Pre]
    BeamformerAudioProcessor* ourProcessor = getProcessor();
    //[/UserslidervalueChanged_Pre]

    if (sliderThatWasMoved == angleSld)
    {
        //[UserSliderCode_angleSld] -- add your slider handling code here..
        ourProcessor->setParameter(BeamformerAudioProcessor::Angle, (float)angleSld->getValue());
        //[/UserSliderCode_angleSld]
    }
    else if (sliderThatWasMoved == spacingSld)
    {
        //[UserSliderCode_spacingSld] -- add your slider handling code here..
        ourProcessor->setParameter(BeamformerAudioProcessor::Spacing, (float)spacingSld->getValue());
        //[/UserSliderCode_spacingSld]
    }
    else if (sliderThatWasMoved == nofmicsSldr)
    {
        //[UserSliderCode_nofmicsSldr] -- add your slider handling code here..
        ourProcessor->setParameter(BeamformerAudioProcessor::Nofmics, (int)nofmicsSldr->getValue());
        //[/UserSliderCode_nofmicsSldr]
    }

    //[UserslidervalueChanged_Post]
    //[/UserslidervalueChanged_Post]
}

void BeamformerAudioProcessorEditor::buttonClicked (Button* buttonThatWasClicked)
{
    //[UserbuttonClicked_Pre]
    BeamformerAudioProcessor* ourProcessor = getProcessor();
    //[/UserbuttonClicked_Pre]

    if (buttonThatWasClicked == BypassBtn)
    {
        //[UserButtonCode_BypassBtn] -- add your button handler code here..
        ourProcessor->setParameterNotifyingHost(BeamformerAudioProcessor::MasterBypass, (float)BypassBtn->getToggleState());
        //[/UserButtonCode_BypassBtn]
    }

    //[UserbuttonClicked_Post]
    //[/UserbuttonClicked_Post]
}
```

Figure 12. Connection between GUI and processing – sending values from sliders

5.3 Processing

When it comes to actual signal processing, even though JUCE makes implementation easier, there were still some challenges to be resolved. An important feature is multiple input channels support, which was realized through taking all possible 15 channels by default and then switching between processing variants depending on user defined parameter.The beamformer implementation can be divided in two stages, that is delaying and summing. They are realized separately but very much linked with each other. In the summing part, all that is done is obtaining values assigned to channels of interest (and representing sound pressure at a given instant of time and point of space), summing them up and dividing the result by the number of them, all of which leads to a quantity known as arithmetic mean. Realization is shown in Figure 13.

```
#include "Beamformer.h"

Beamformer::Beamformer() { SetAngle(0.0f); SetSpacing(0.005f); SetNofmics(3); }
Beamformer::~Beamformer(){}

void Beamformer::SetAngle(float angle)
{
    m_angle=angle;
}
void Beamformer::SetSpacing(float spacing)
{
    m_spacing = spacing;
}
void Beamformer::SetNofmics(int nofmics)
{
    m_nofmics = nofmics;
}

void Beamformer::ClockProcess(int nofmics, float* LeftSample, float* RightSample, float* ThirdSample, float* FourthSample, floa
{
    float m = 0;
    switch (nofmics)
    {
    case 3:
        m = (*LeftSample + *RightSample + *ThirdSample) / 3;
        break;
    case 5:
        m = (*LeftSample + *RightSample + *ThirdSample + *FourthSample + *FifthSample) / 5;
        break;
    case 7:
        m = (*LeftSample + *RightSample + *ThirdSample + *FourthSample + *FifthSample + *SixthSample + *SeventhSample) / 7;
        break;
    case 9:
        m = (*LeftSample + *RightSample + *ThirdSample + *FourthSample + *FifthSample + *SixthSample + *SeventhSample + *Eighth
        break;
    case 11:
        m = (*LeftSample + *RightSample + *ThirdSample + *FourthSample + *FifthSample + *SixthSample + *SeventhSample + *Eighth
        break;
    case 13:
        m = (*LeftSample + *RightSample + *ThirdSample + *FourthSample + *FifthSample + *SixthSample + *SeventhSample + *Eighth
        break;
    case 15:
        m = (*LeftSample + *RightSample + *ThirdSample + *FourthSample + *FifthSample + *SixthSample + *SeventhSample + *Eighth
        break;
    default:
        m = 0.0f;
        break;
    }
*LeftSample=m;
*RightSample=m;
}
```

Figure 13. Summation of the called channels

The calculated value is later written to both left and right output channels, so even though the plugin produces a stereo output, technically it is a duplicated monophonic one.

Delaying, the second processing stage, is performed while calling the above described summing function, simply by first computing the amount of delay (in samples) to be applied and then adding or subtracting that amount or its multiple (depending on the number of microphones involved) from channel indexes. The code embodies an approach of perserving the center microphone to be the reference one at all times and is presented in Figure 14.

```
int BeamformerAudioProcessor::samplesDelay(int n)
{
    double res = sqrt(1 + (2 * n * UserParams[Spacing] * cos(UserParams[Angle]) + pow(n * UserParams[Spacing], 2)))/330;
    return (int)ceil(res*getSampleRate());
}

void BeamformerAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
{
    if(getNumInputChannels()<2 || UserParams[MasterBypass])
    {}/*Nothing to do here - processing is in-place, so doing nothing is pass-through (for NumInputs=NumOutputs)*/
    else
    {//Not bypassed - do processing!
        float* leftData = buffer.getWritePointer(0);
        float* rightData = buffer.getWritePointer(1);
        float* thirdData = buffer.getWritePointer(2);
        float* fourthData = buffer.getWritePointer(3);
        float* fifthData = buffer.getWritePointer(4);
        float* sixthData = buffer.getWritePointer(5);
        float* seventhData = buffer.getWritePointer(6);
        float* eigthData = buffer.getWritePointer(7);
        float* ninthData = buffer.getWritePointer(8);
        float* tenthData = buffer.getWritePointer(9);
        float* eleventhData = buffer.getWritePointer(10);
        float* twelvethData = buffer.getWritePointer(11);
        float* thirteenthData = buffer.getWritePointer(12);
        float* fourteenthData = buffer.getWritePointer(13);
        float* fifteenthData = buffer.getWritePointer(14);

        //float secondsDelay = UserParams[Spacing] * sin(UserParams[Angle]) / 330;
        //int samplesDelay = ceil(secondsDelay*getSampleRate());
        //int samplesDelay = ceil(secondsDelay*getSampleRate());
        switch ((int)UserParams[Nofmics])
        {
        case 3:
            for (long i = samplesDelay(1); i < (buffer.getNumSamples() - samplesDelay(1)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(1)], &rightData[i], &thirdData[i + sa
            break;
        case 5:
            for (long i = samplesDelay(2); i < (buffer.getNumSamples() - samplesDelay(2)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(2)], &rightData[i - samplesDelay(1)],
            break;
        case 7:
            for (long i = samplesDelay(3); i < (buffer.getNumSamples() - samplesDelay(3)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(3)], &rightData[i - samplesDelay(2)],
            break;
        case 9:
            for (long i = samplesDelay(4); i < (buffer.getNumSamples() - samplesDelay(4)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(4)], &rightData[i - samplesDelay(3)],
            break;
        case 11:
            for (long i = samplesDelay(5); i < (buffer.getNumSamples() - samplesDelay(5)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(5)], &rightData[i - samplesDelay(4)],
            break;
        case 13:
            for (long i = samplesDelay(6); i < (buffer.getNumSamples() - samplesDelay(6)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(6)], &rightData[i - samplesDelay(5)],
            break;
        case 15:
            for (long i = samplesDelay(7); i < (buffer.getNumSamples() - samplesDelay(7)); i++)
                mBeamformer.ClockProcess((int)UserParams[Nofmics], &leftData[i - samplesDelay(7)], &rightData[i - samplesDelay(6)],
            break;
        default:
            break;
        }
    }
}
```

Figure 14. A function is called with delayed sample indexes

For debugging and finally running the plugin a host must be used that is capable of supporting multiple inputs and a stereo output. Typically, majority of DAWs can utilize only stereo input and output, some of them are only monophonic. This is due to the fact that for most application in music production, this is sufficient, as most music lovers still tend to prefer to listen to two channels over engaging in immersive, spatial sound systems. Possibly and hopefully this trend will gradually shift, but as for this moment a special purpose VST host must have been found. A brief search led the author to Audio Mulch [47], a flexible tool for sound design and music production, that's incredibly easy to operate with and offered free of charge for 60 days trial period. Inside Audio Mulch, the debugging setup was as simple as it can be seen in Figure 15.

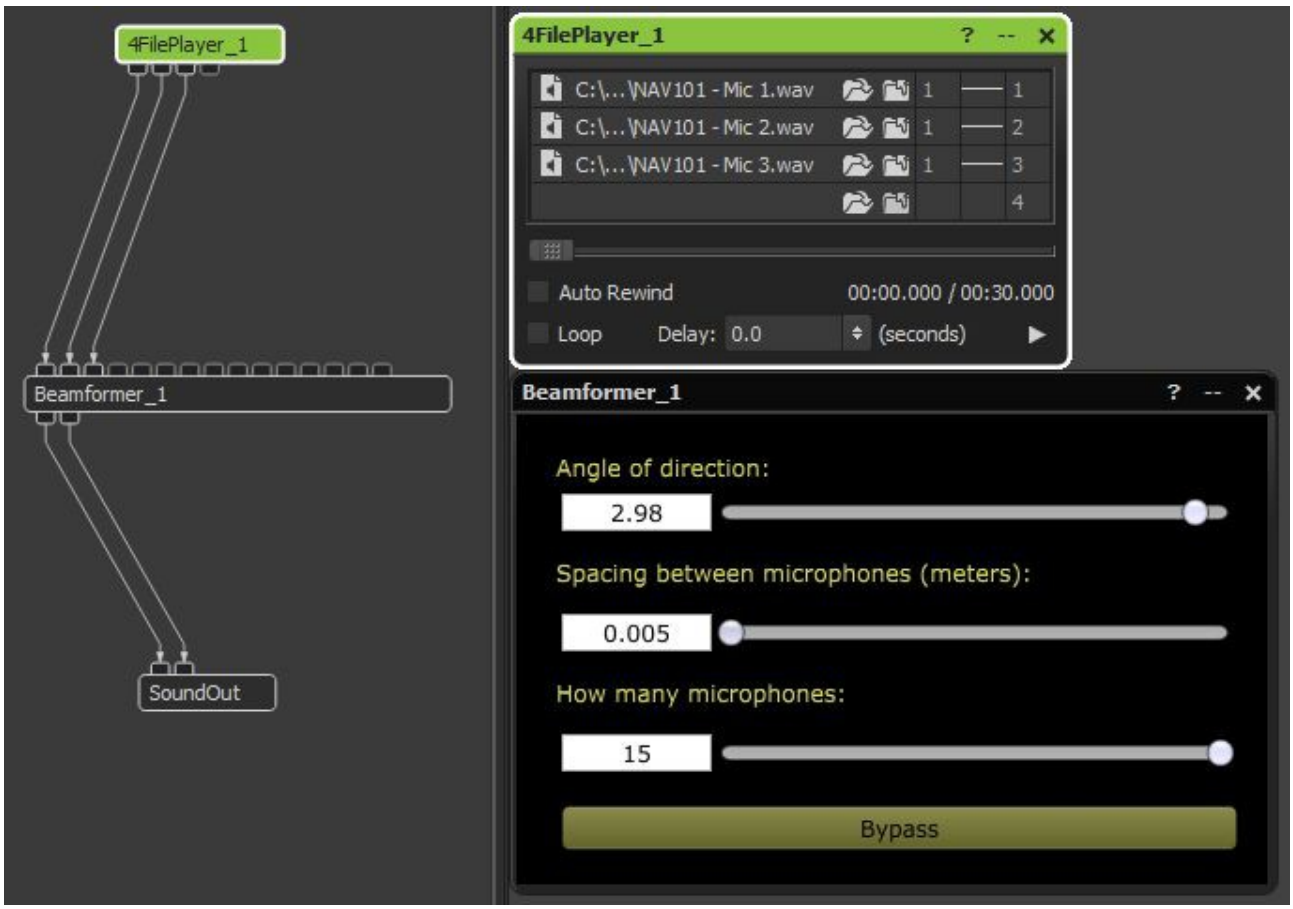


Figure 15. VST debugging environment in Audio Mulch software

6. Evaluation

Throughout prototyping, designing, debuggin and finally evaluation of the project, a prerecorded sound tracks have been used as a groundwork. They have been mentioned in previous chapters and here a detailed description of their acquisition will be presented. On top of that both pre- and post-array spectral densities, array gain and empiric evaluation will be conducted and lead to a conclusion and future perspectives, forming the following, final chapter.

6.1 Recording of the sample tracks

The recordings were done at Royal Danish Academy of Music, thankfully to Jesper Andersen, head of Tonmeister education program. This collaboration was important from recording conditions and equipment perspective - the author was granted access to a large and properly set up recording studio, with an outstandingly diffuse reverberation and negligibly little resonance. In addition to that, thankfully to using this facility, top notch microphones have been incorporated into the DIY array arranged for this project's purpose. Precisely speaking, it were three DPA 4006 condenser

microphones, possessing exceptional parameters. According to the manufacturer, they present frequency response that is flat with +/- 2dB tolerance in range 10Hz-20kHz and nearly perfect omnidirectionality in range essential from speech transmission (up to 4kHz) - please refer to Figures 16 and 17 for precise details. Additionally, the microphone's physical dimensions were small enough for a fairly closely-spaced array to be arranged, with inter-center spacing of 5mm (Figure 18). There was also a second series of recordings made, in which sensors were spaced at 20mm. Please note that apart from direct influence on the array's performance, these values are also definitive when calculating spatial aliasing associated with it. In those cases, when inserting spacing values into (16), noting that $f = \frac{c}{\lambda}$, results of 33000Hz and 8250Hz are obtained, with an assumption for speed of sound to be equal to 330 m/s. In other words, frequencies above these values are misrepresented by arrays arranged for this project's recordings.

Another constraint was to keep sound sources in near field of the array. To ensure that, an effective length of aperture was assumed as distance between outermost microphones. For abovementioned spacing values, the borderline was given by 1.88m and 4.86m for 0.005m and 0.02m respectively. Hence for simplicity of setup it was decided to keep sources 1m away from the array at all times.

As sound sources, a pair of medium sized studio monitors Dynaudio BM15A was used. They exhibit very high sound quality with frequency range between 40Hz and 22kHz, maximum sound pressure level of 124dB RMS and dimensions small enough to be used in near field. The final piece of signal chain, serving as both preamp and analog-to-digital converter for microphones and digital-to-analog converter feeding the speakers, was RME Fireface 800 audio interface.

Recordings were made in five different relative sources positions, presented in Figure 19. Unfortunately, the author did not decide to record one source on the end-fire (alongside) axis with the other on the broadside (athwart one) and he regrets it. Supposedly, in such configuration the array's impact on sound sources reception would be most apparent.

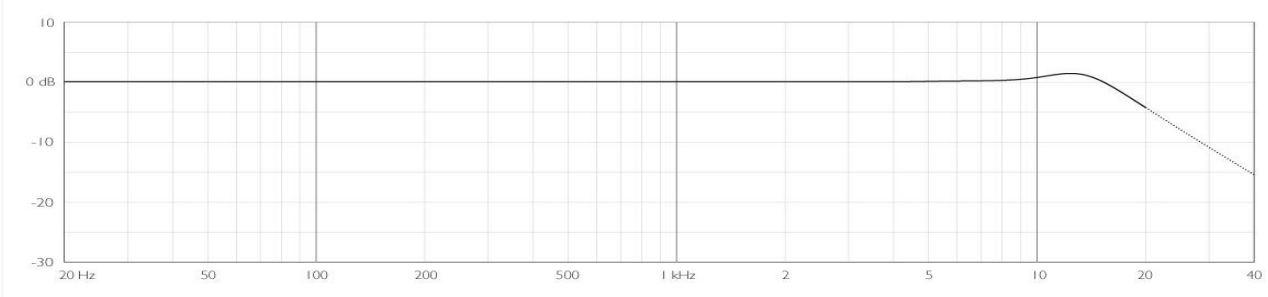


Figure 16. Frequency response of a DPA4006 microphone

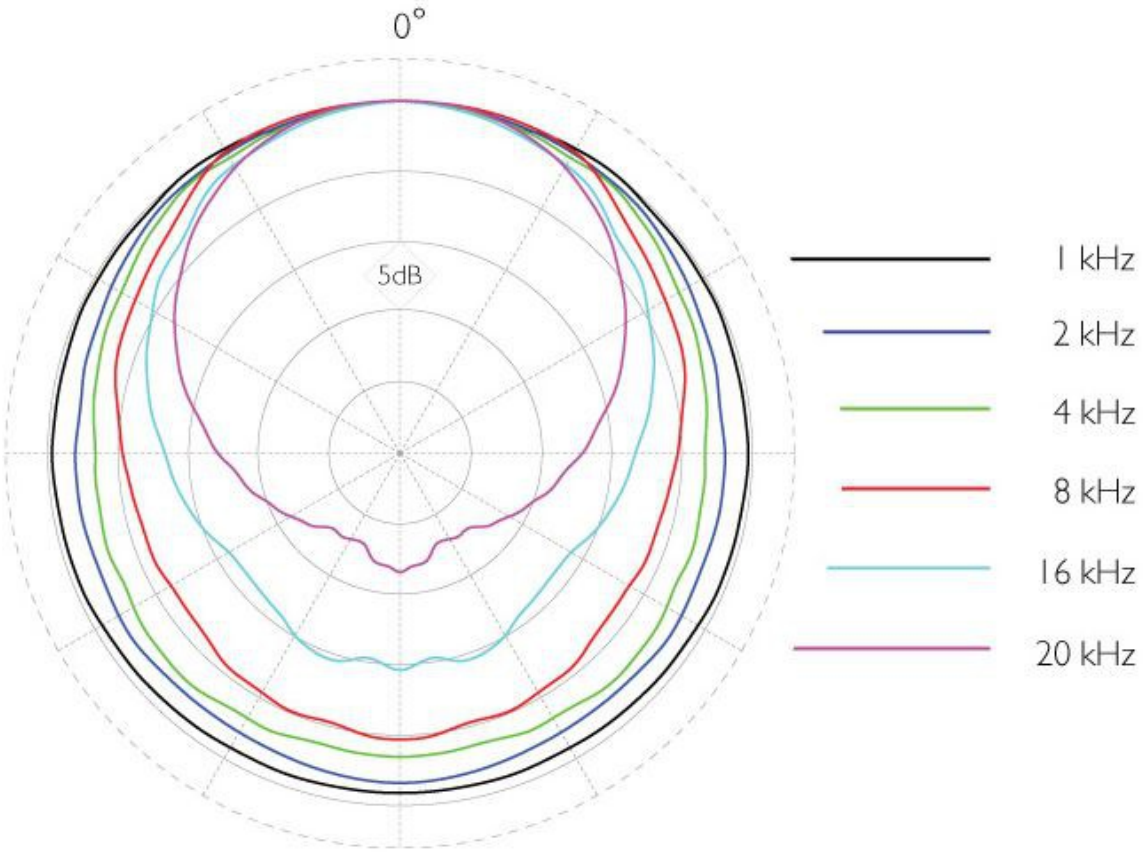


Figure 17. Directivity pattern of a DPA4006 microphone

Figure 18. Closest possible spacing with DPA4006 microphones

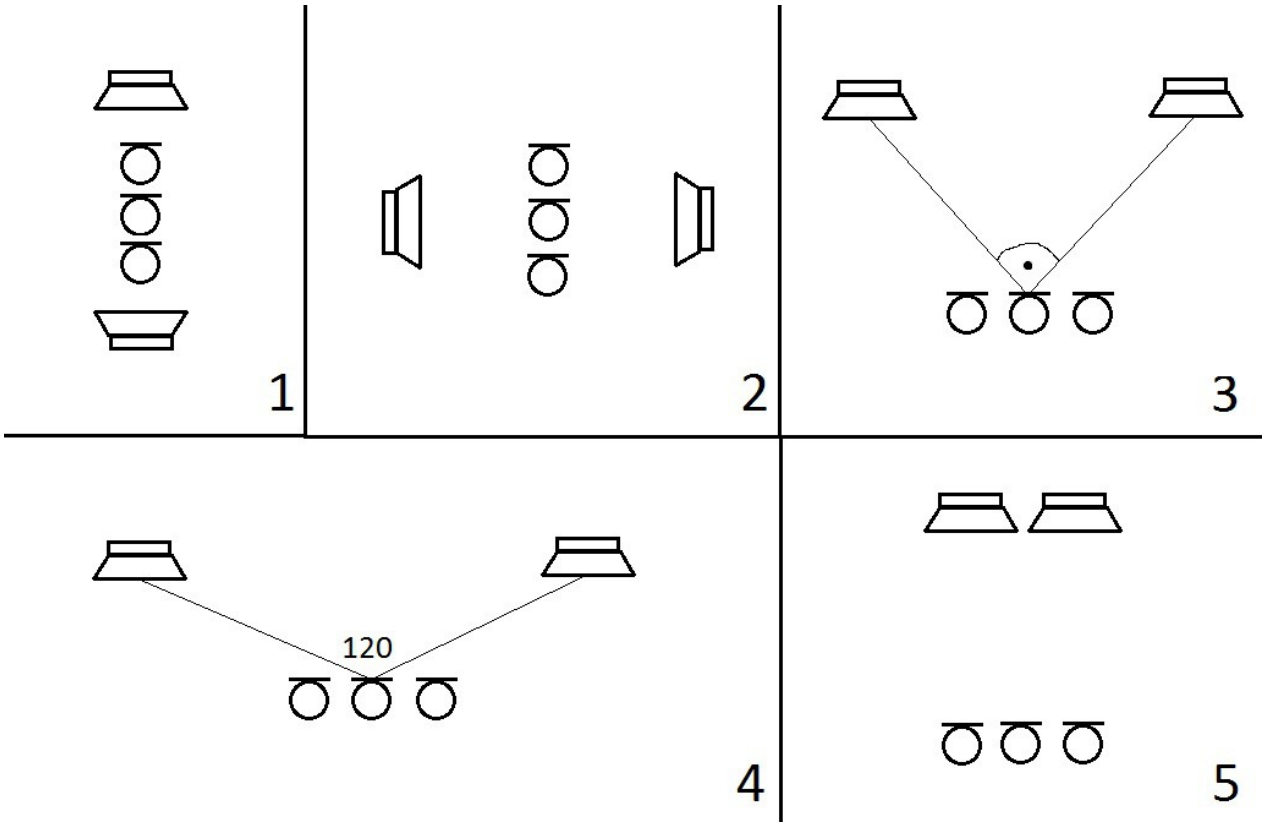


Figure 19. Positions of sound sources schematic

Finally, twelve mixes of six different sound types were used in order to evaluate the array's performance in various scenarios. These included:

1. Male voice in one speaker and female in the other
2. Male voice in one speaker and piano in the other
3. Male voice in one speaker and guitar in the other
4. Male voice in one speaker and pure tone of frequency 300Hz in the other
5. Male voice in one speaker and pure tone of frequency 3000Hz in the other
6. Female voice in one speaker and piano in the other
7. Female voice in one speaker and guitar in the other
8. Female voice in one speaker and pure tone of frequency 300Hz in the other
9. Female voice in one speaker and pure tone of frequency 3000Hz in the other
10. Piano in one speaker and guitar in the other
11. Piano in one speaker and pure tone of frequency 300Hz in the other
12. Guitar in one speaker and pure tone of frequency 300Hz in the other

Sound recordings used to feed the speakers come from various origins. Speech excerpts come from

a database contributed by University of Edinburgh's Centre for Speech Technology Research, known as the Edinburgh University Speech Timing Archive and Corpus of English (EUSTACE) [48]. The piano music snippet was taken from Frederic Chopin's Prelude in F Sharp Minor Op. 28, played by Maurizio Pollini. The guitar sample is Antonio Lauro's Seis por Derecho, performed by John Williams. Pure tones were generated using Audacity workstation.

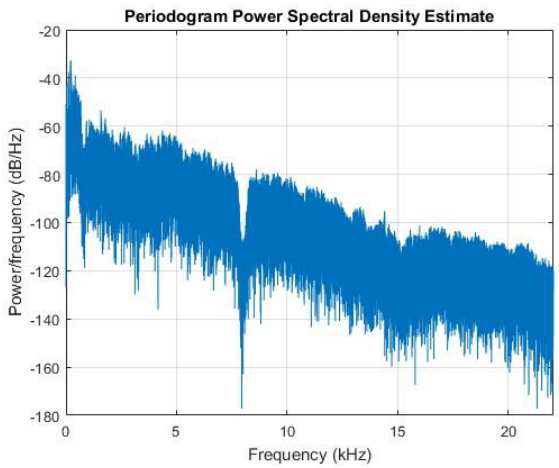
A total of 120 recordings were acquired, each consisting of three tracks (one for each microphone in the array). These sound files were extensively used in prototyping and debugging of the plugin and most relevant of them were picked for evaluation presented in the following subchapter.

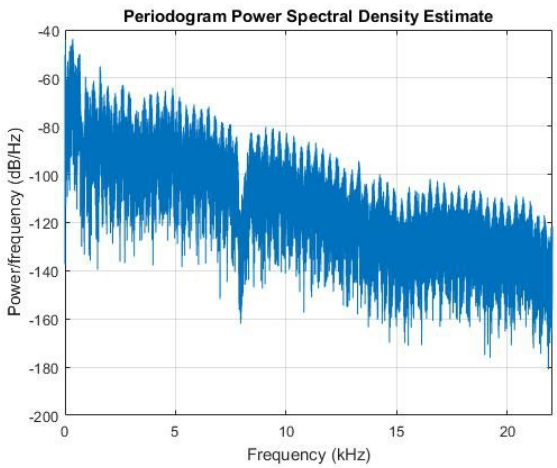
6.2 Evaluation of the array performance

For evaluation of the array performance, there was only one spatial condition chosen, number 3 as seen in Figure 19. It was due to the fact that in such configuration, with sound sources located on perpendicular axes, spatial filtering should be most noticeable. Additionally, a condition of two voices speaking simultaneously in all recorded positions is included, as this case is most important from practical point of view. Furthermore, a distinction is made between two spacing conditions. This evaluation is concluded with an empirical analysis of results and resulting sound files are included as media attachment for this thesis when handed in.

Unprocessed Processed

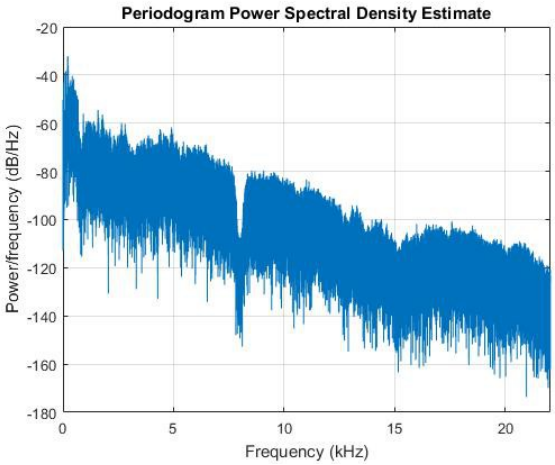
1. End-fire axis, d=0.02m, speech in both speakers



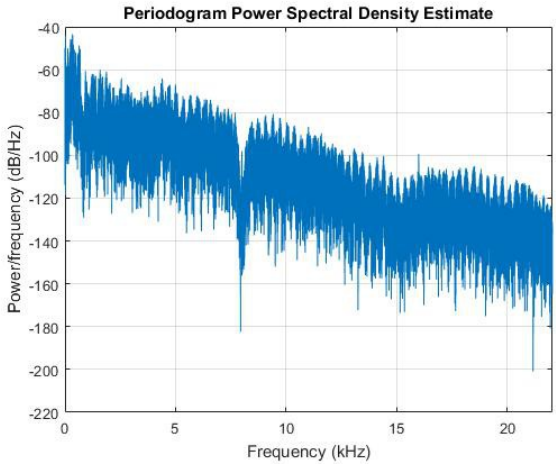


Average Power/Frequency = 1.88 * 10^(-3) Average Power/Frequency = 0.28 * 10^(-3)

2. Broadside axis, d=0.02m, speech in both speakers

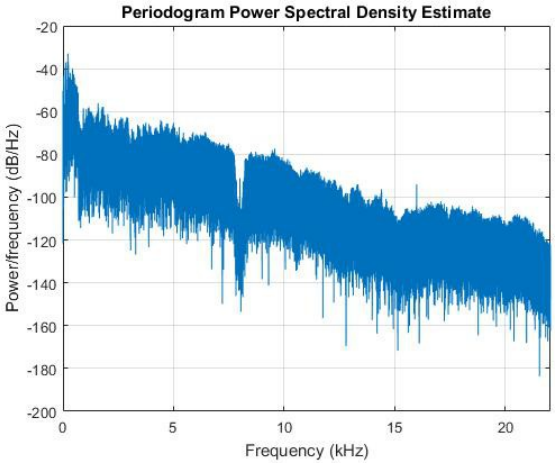


Average Power/Frequency = 2.10 * 10^(-3)

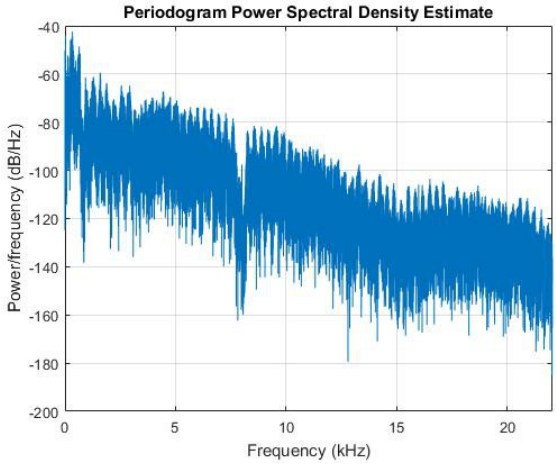


Average Power/Frequency = 0.31 * 10^(-3)

3. Sources on perpendicular axes, d=0.02m, speech in both speakers

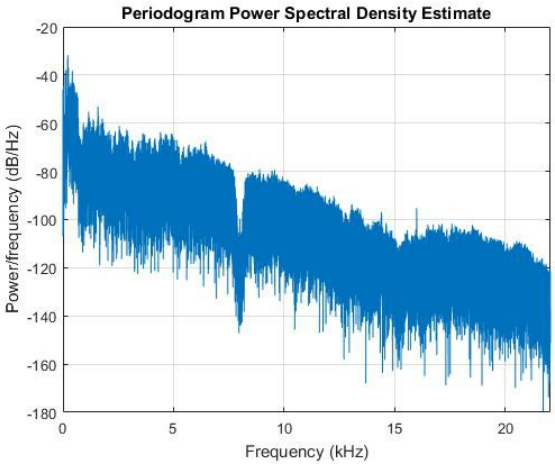


Average Power/Frequency = 1.76 * 10^(-3)

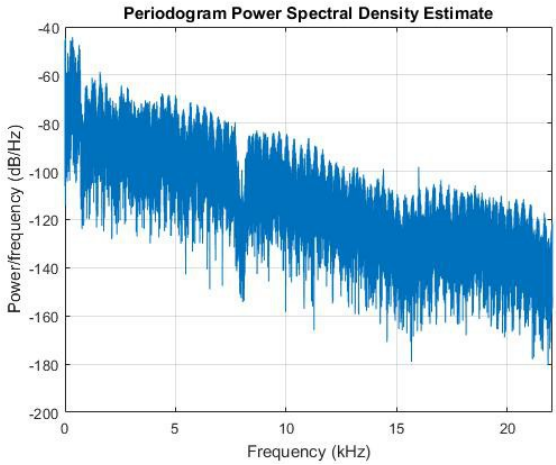


Average Power/Frequency = 0.29 * 10^(-3)

4. Sources on axes forming a 120 degrees angle, d=0.02m, speech in both speakers

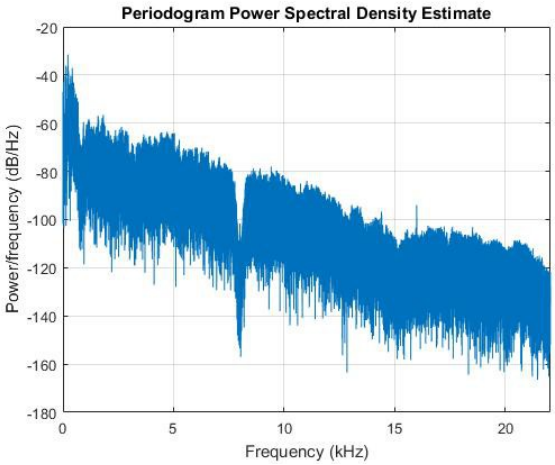


Average Power/Frequency = 2.11 * 10^(-3)

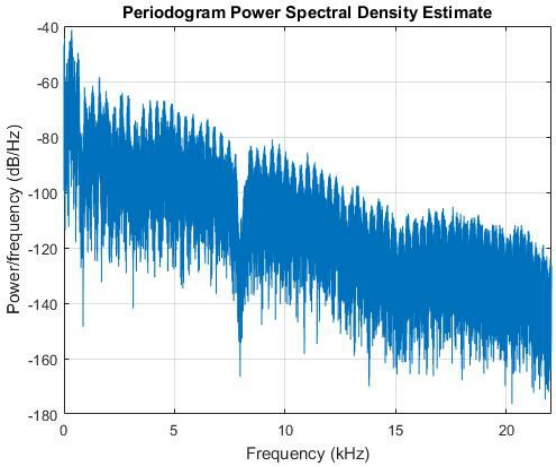


Average Power/Frequency = 0.31 * 10^(-3)

5. Sources next to each other on the broadside axis, d=0.02m, speech in both speakers

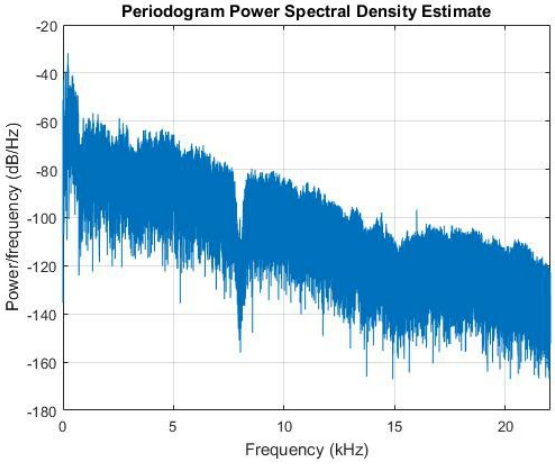


Average Power/Frequency = 2.17 * 10^(-3)

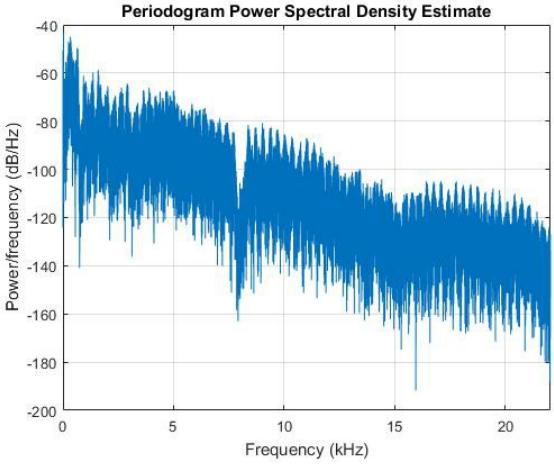


Average Power/Frequency = 0.35 * 10^(-3)

6. End-fire axis, d=0.005m, speech in both speakers

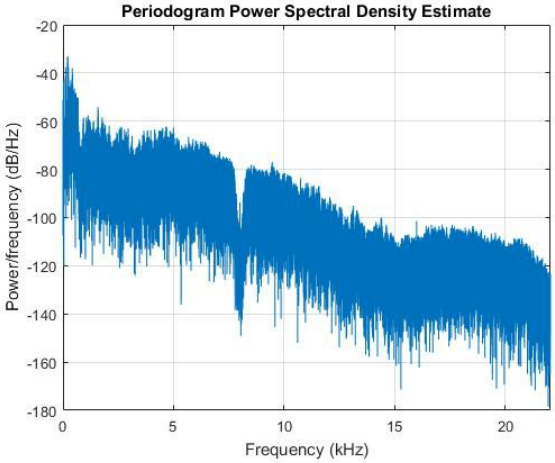


Average Power/Frequency = 1.72 * 10^(-3)

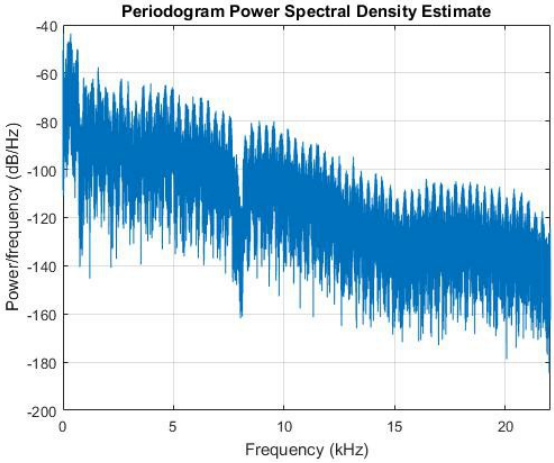


Average Power/Frequency = 0.22 * 10^(-3)

7. Broadside axis, d=0.005m, speech in both speakers

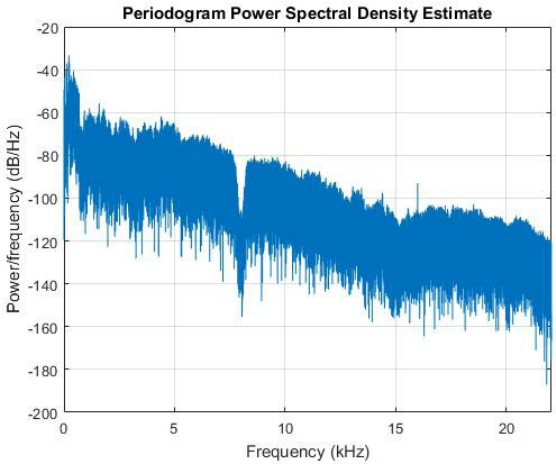


Average Power/Frequency = 1.76 * 10^(-3)

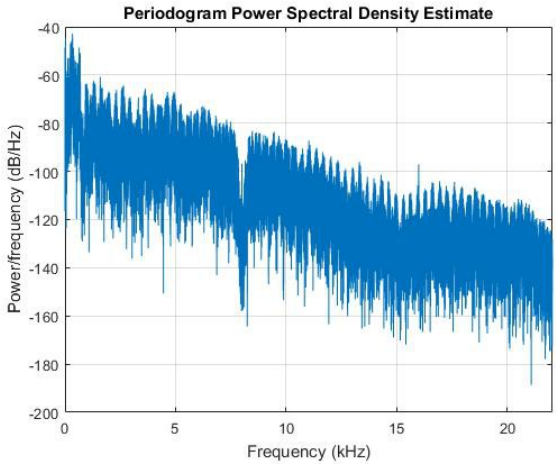


Average Power/Frequency = 0.24 * 10^(-3)

8. Sources on perpendicular axes, d=0.005m, speech in both speakers

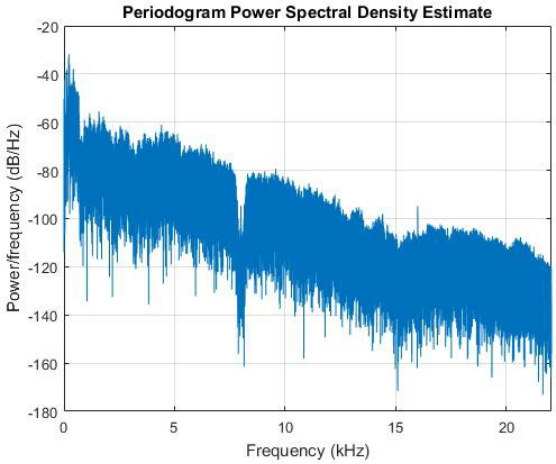


Average Power/Frequency = 1.70 * 10^(-3)

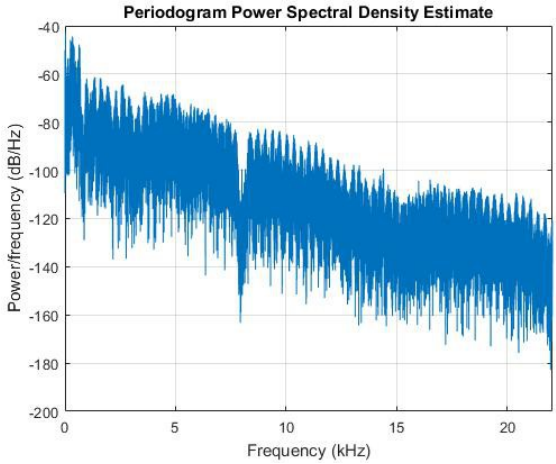


Average Power/Frequency = 0.26 * 10^(-3)

9. Sources on axes forming a 120 degrees angle, d=0.005m, speech in both speakers

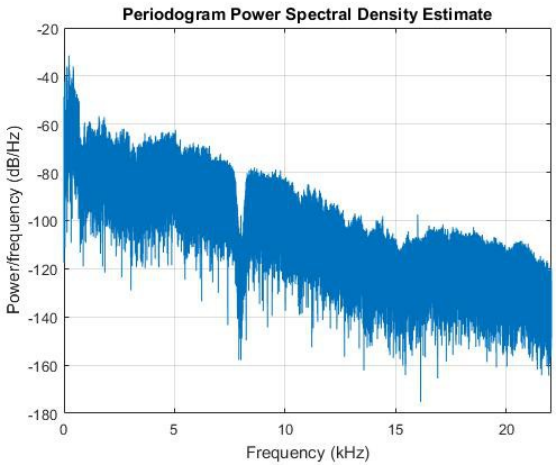


Average Power/Frequency = 2.21 * 10^(-3)

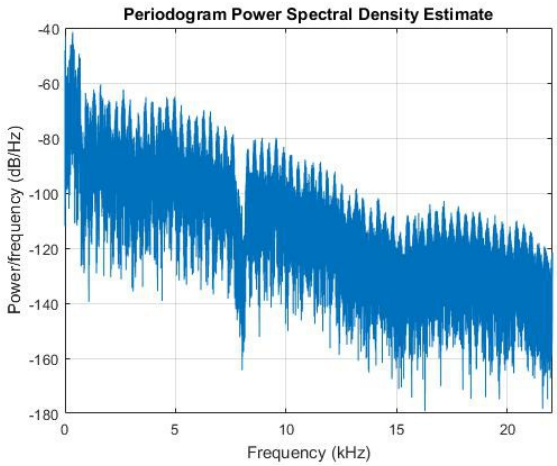


Average Power/Frequency = 0.28 * 10^(-3)

10. Sources next to each other on the broadside axis, d=0.005m, speech in both speakers

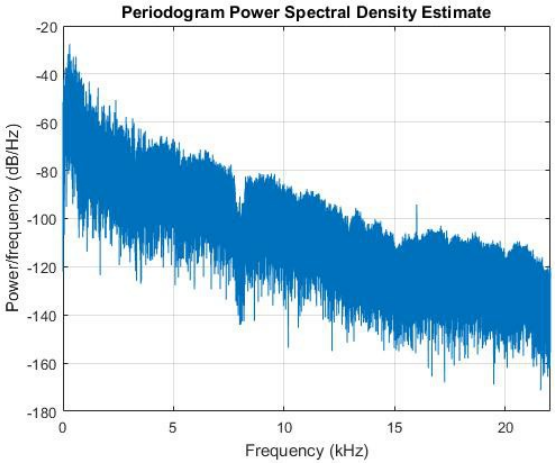


Average Power/Frequency = 2.26 * 10^(-3)

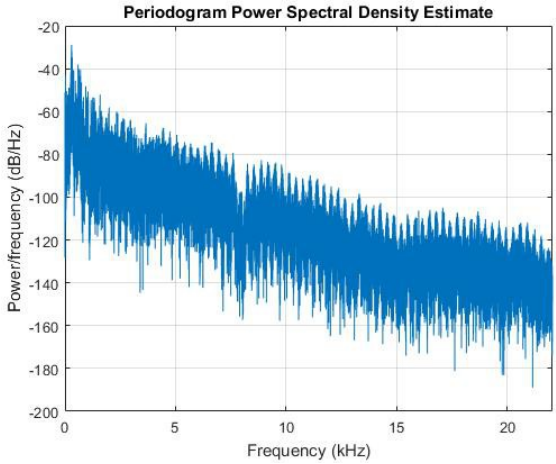


Average Power/Frequency = 0.33 * 10^(-3)

11. Sources on perpendicular axes, d=0.02m, male voice and piano

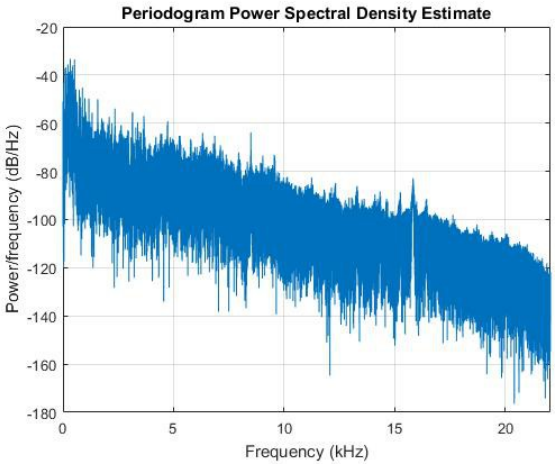


Average Power/Frequency = 2.69 * 10^(-3)

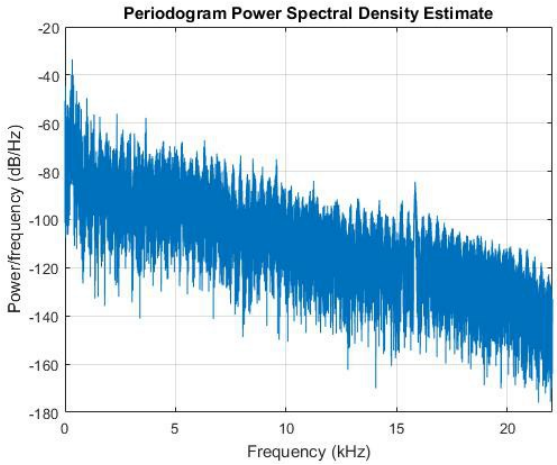


Average Power/Frequency = 0.79 * 10^(-3)

12. Sources on perpendicular axes, d=0.02m, male voice and guitar

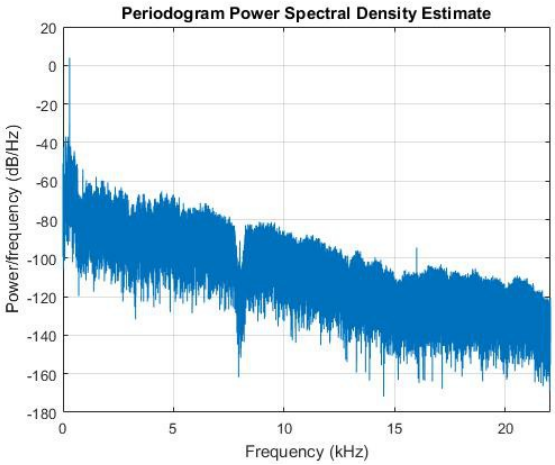


Average Power/Frequency = 0.40 * 10^(-3)

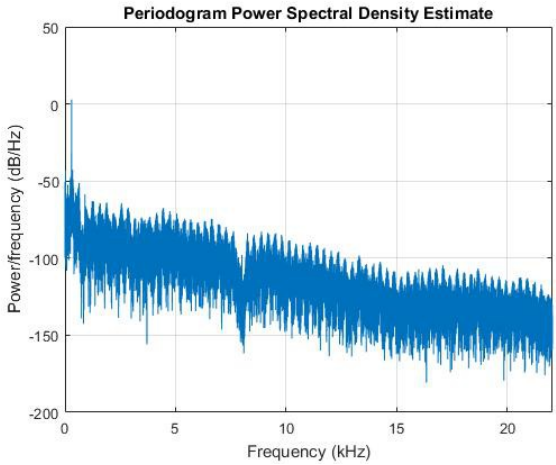


Average Power/Frequency = 0.17 * 10^(-3)

13. Sources on perpendicular axes, d=0.02m, male voice and 300Hz

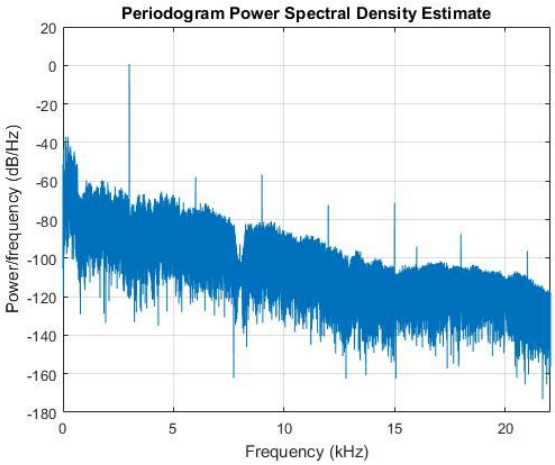


Average Power/Frequency = 37.12 * 10^(-3)

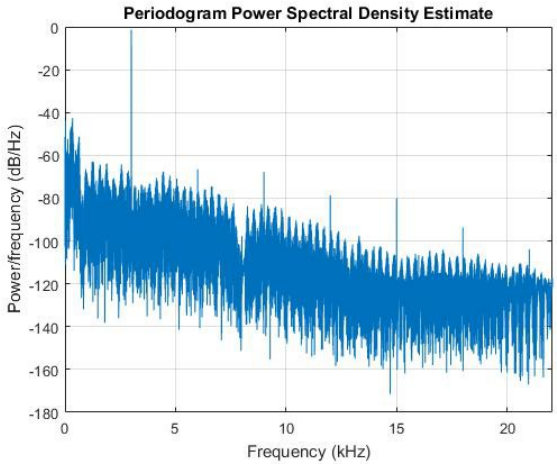


Average Power/Frequency = 28.74 * 10^(-3)

14. Sources on perpendicular axes, d=0.02m, male voice and 3000Hz

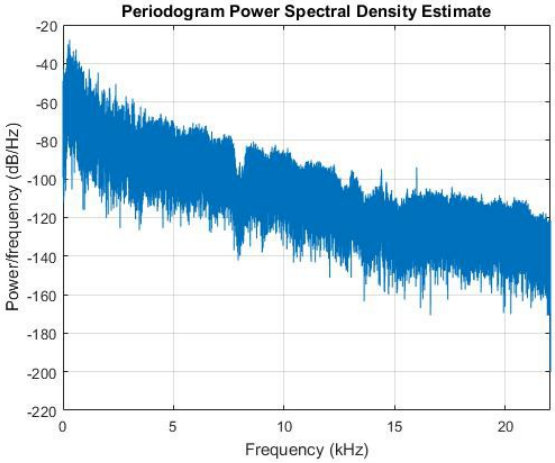


Average Power/Frequency = 17.89 * 10^(-3)

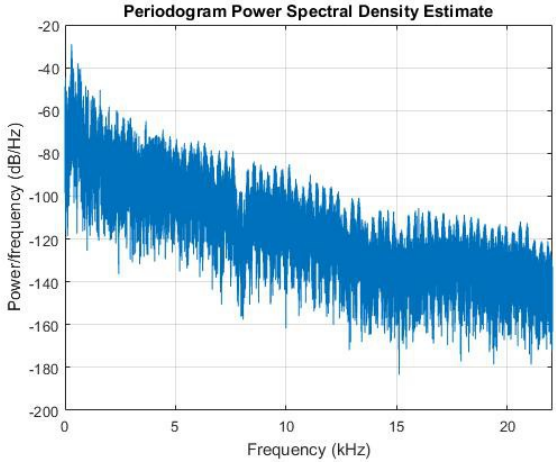


Average Power/Frequency = 11.85 * 10^(-3)

15. Sources on perpendicular axes, d=0.02m, female voice and piano

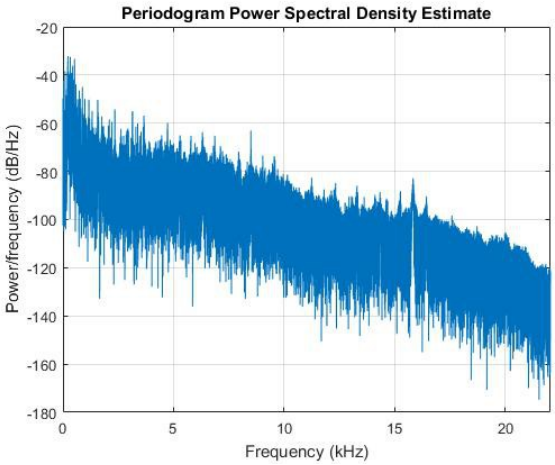


Average Power/Frequency = 2.6 * 10^(-3)

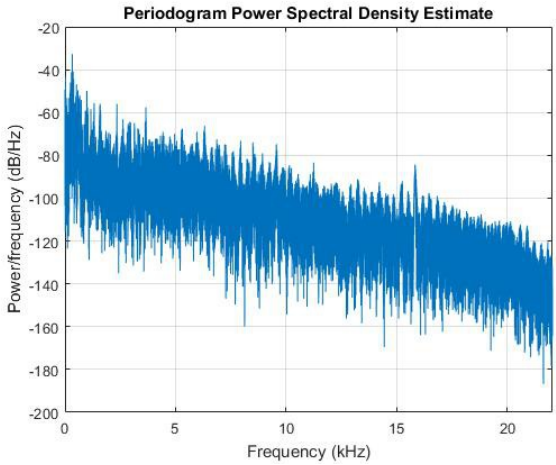


Average Power/Frequency = 0.67 * 10^(-3)

16. Sources on perpendicular axes, d=0.02m, female voice and guitar

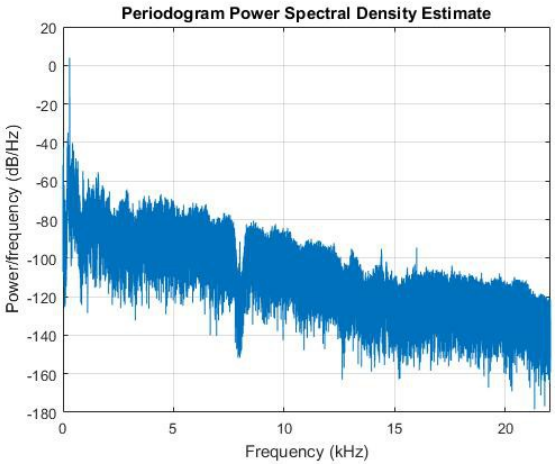


Average Power/Frequency = 1.61 * 10^(-3)

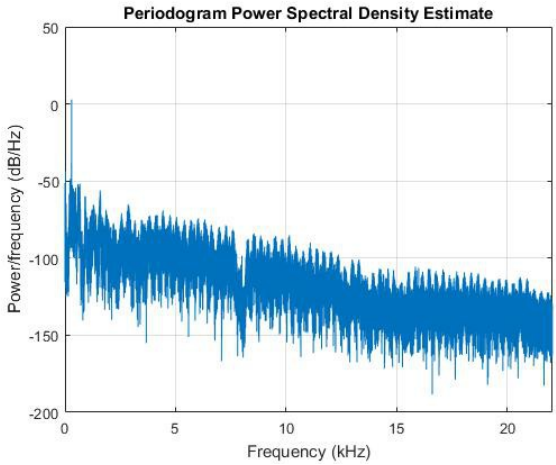


Average Power/Frequency = 0.29 * 10^(-3)

17. Sources on perpendicular axes, d=0.02m, female voice and 300Hz

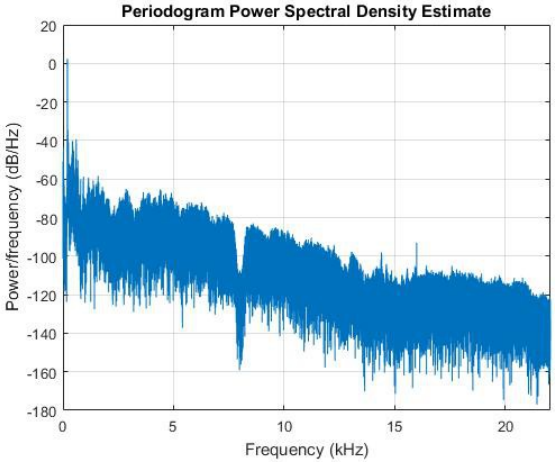


Average Power/Frequency = 36.98 * 10^(-3)

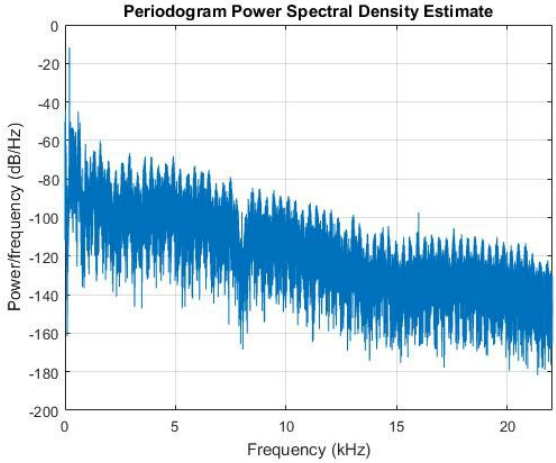


Average Power/Frequency = 28.57 * 10^(-3)

18. Sources on perpendicular axes, d=0.02m, female voice and 3000Hz

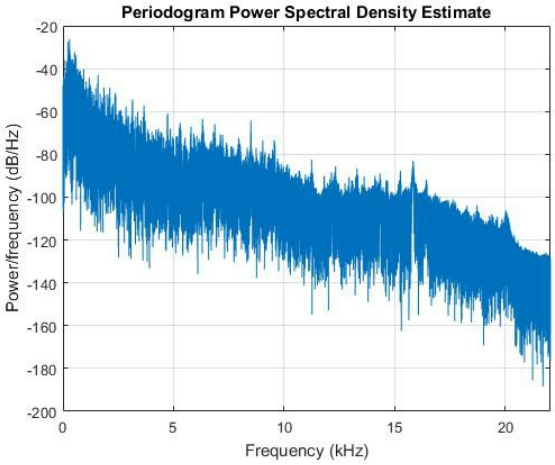


Average Power/Frequency = 20.12 * 10^(-3)

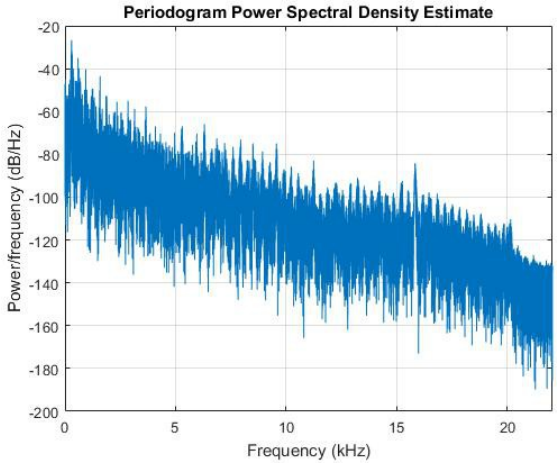


Average Power/Frequency = 13.13 * 10^(-3)

19. Sources on perpendicular axes, d=0.02m, piano and guitar

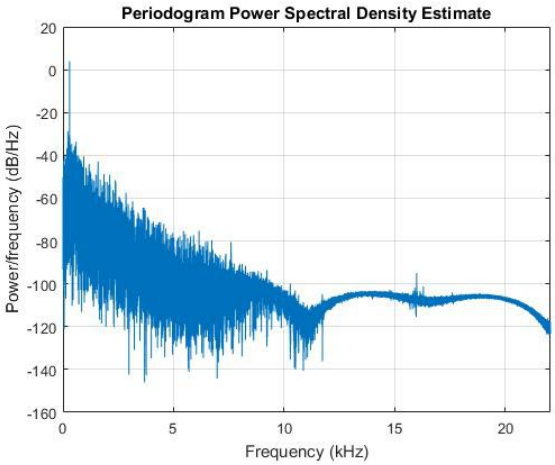


Average Power/Frequency = 2.64 * 10^(-3)

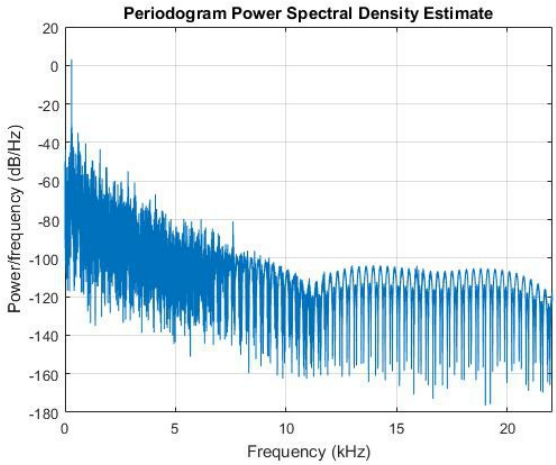


Average Power/Frequency = 1.04 * 10^(-3)

20. Sources on perpendicular axes, d=0.02m, piano and 300Hz

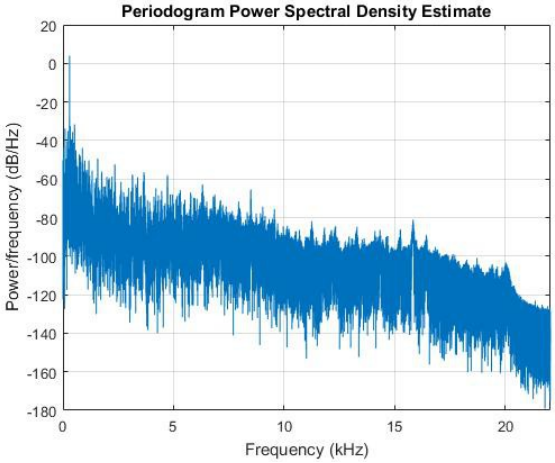


Average Power/Frequency = 37.8 * 10^(-3)

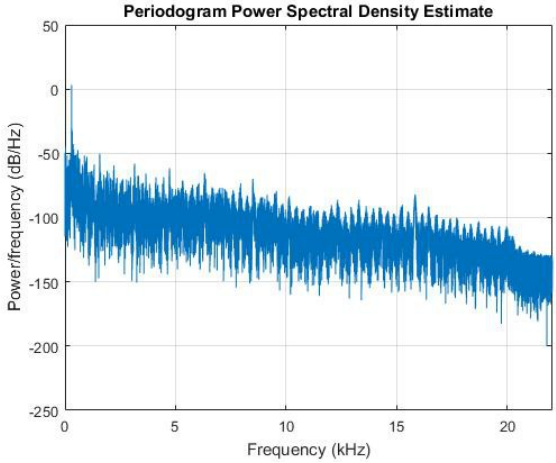


Average Power/Frequency = 31.1 * 10^(-3)

21. Sources on perpendicular axes, d=0.02m, guitar and 300Hz

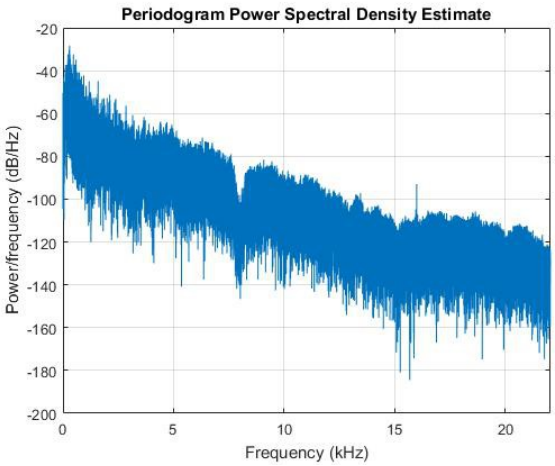


Average Power/Frequency = 36.77 * 10^(-3)

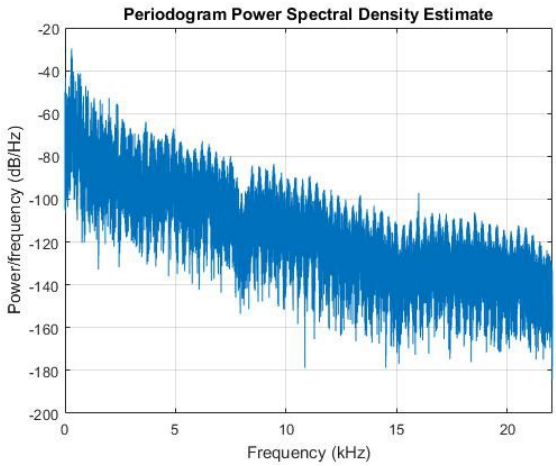


Average Power/Frequency = 30.51 * 10^(-3)

22. Sources on perpendicular axes, d=0.005m, male voice and piano

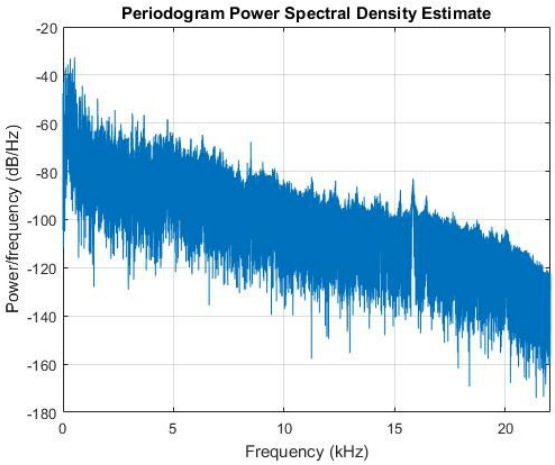


Average Power/Frequency = 2.45 * 10^(-3)

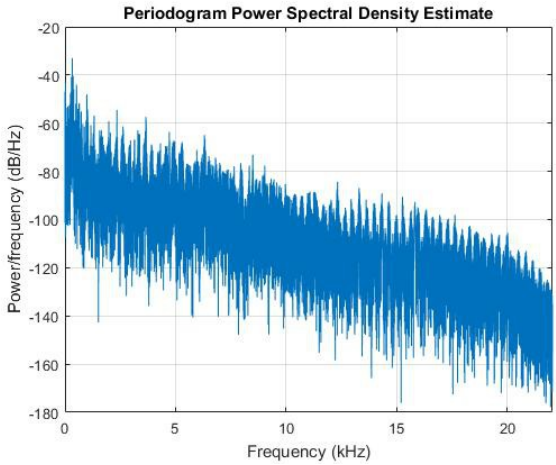


Average Power/Frequency = 0.74 * 10^(-3)

23. Sources on perpendicular axes, d=0.005m, male voice and guitar

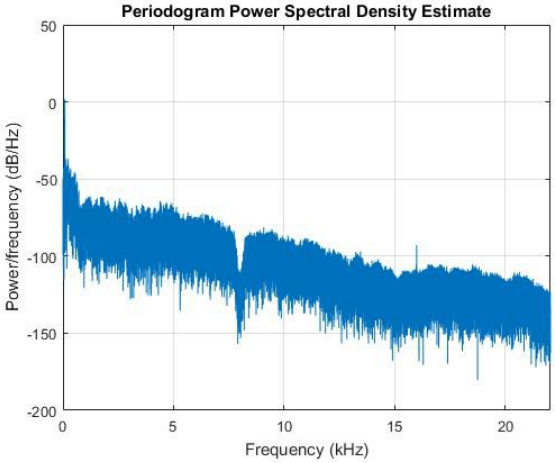


Average Power/Frequency = 1.56 * 10^(-3)

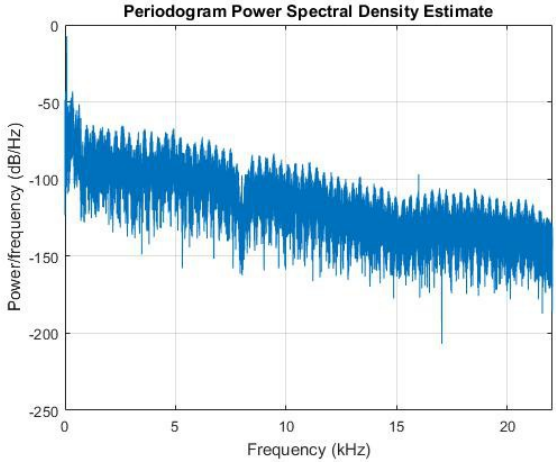


Average Power/Frequency = 0.38 * 10^(-3)

24. Sources on perpendicular axes, d=0.005m, male voice and 300Hz

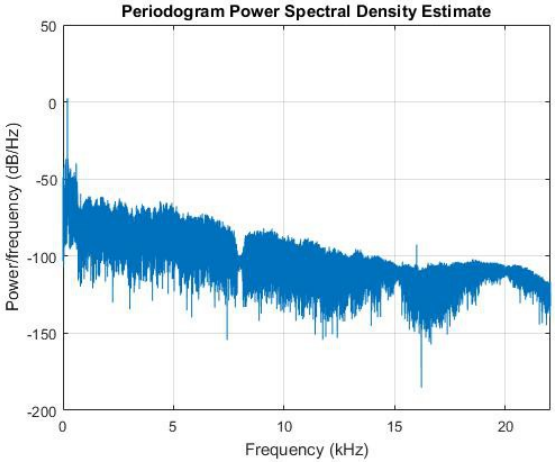


Average Power/Frequency = 22.35 * 10^(-3)

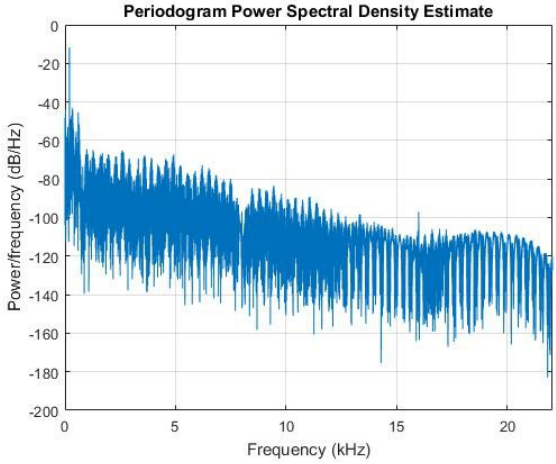


Average Power/Frequency = 3.16 * 10^(-3)

25. Sources on perpendicular axes, d=0.005m, male voice and 3000Hz

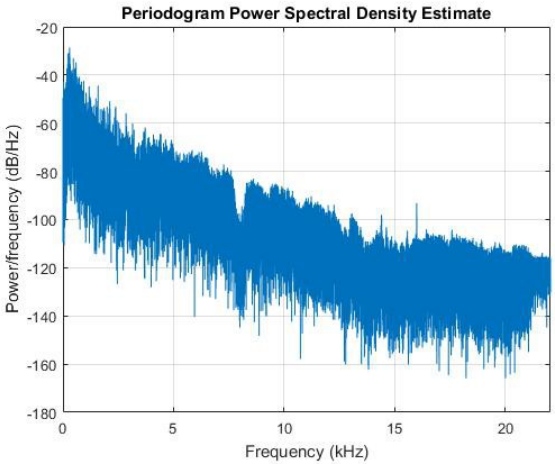


Average Power/Frequency = 25.56 * 10^(-3)

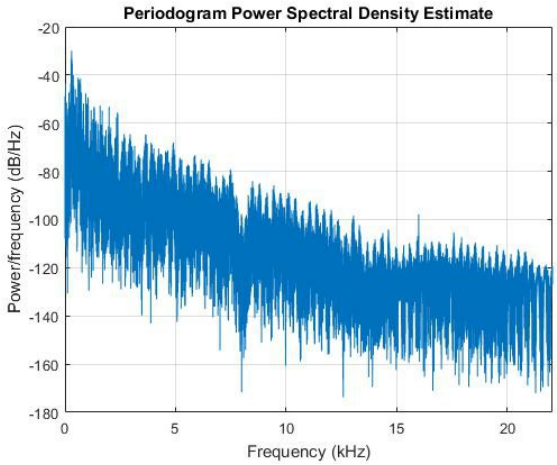


Average Power/Frequency = 1.4 * 10^(-3)

26. Sources on perpendicular axes, d=0.005m, female voice and piano

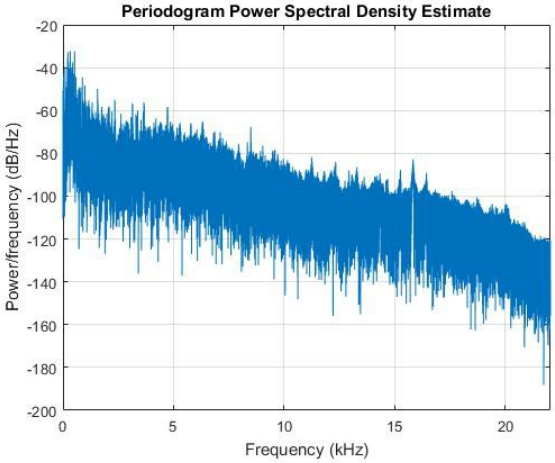


Average Power/Frequency = 2.38 * 10^(-3)

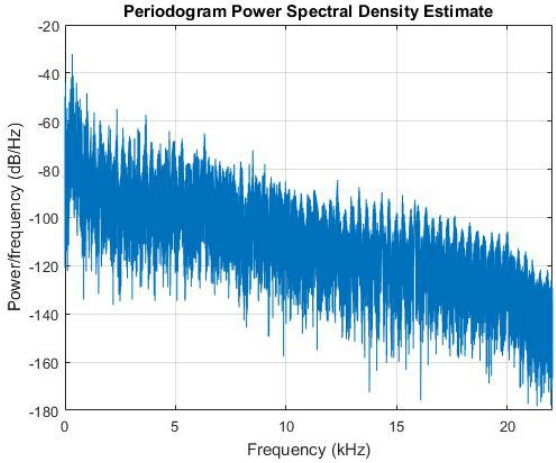


Average Power/Frequency = 0.62 * 10^(-3)

27. Sources on perpendicular axes, d=0.005m, female voice and guitar

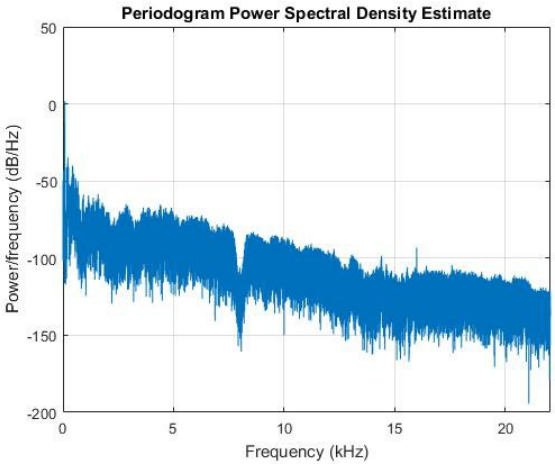


Average Power/Frequency = 1.52 * 10^(-3)

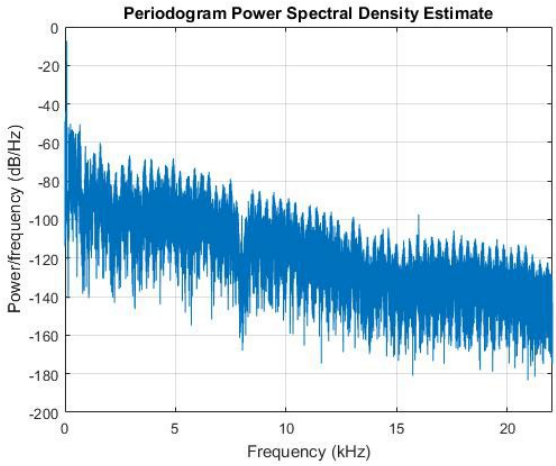


Average Power/Frequency = 0.27 * 10^(-3)

28. Sources on perpendicular axes, d=0.005m, female voice and 300Hz

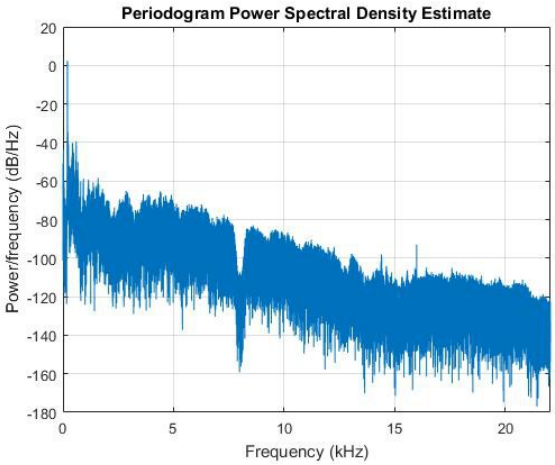


Average Power/Frequency = 22.25 * 10^(-3)

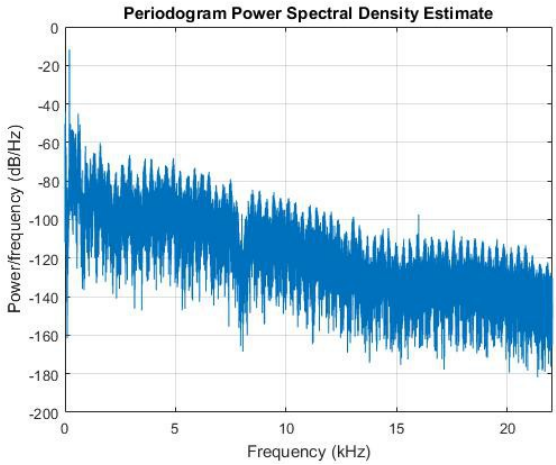


Average Power/Frequency = 3.05 * 10^(-3)

29. Sources on perpendicular axes, d=0.005m, female voice and 3000Hz

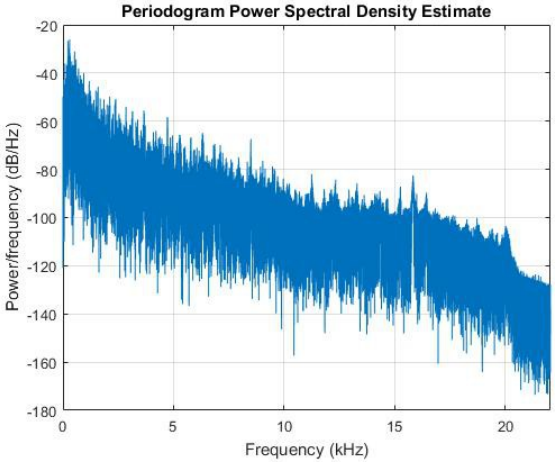


Average Power/Frequency = 25.69 * 10^(-3)

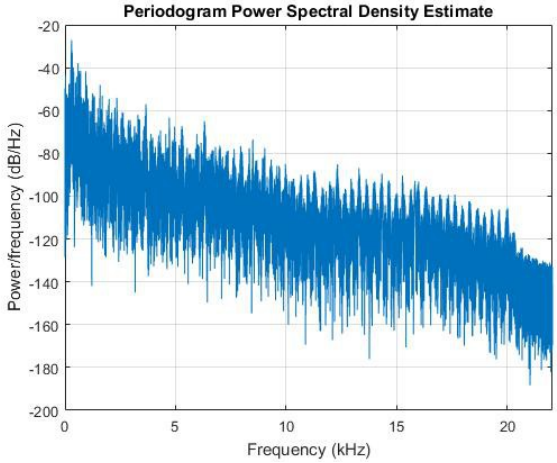


Average Power/Frequency = 1.3 * 10^(-3)

30. Sources on perpendicular axes, d=0.005m, piano and guitar

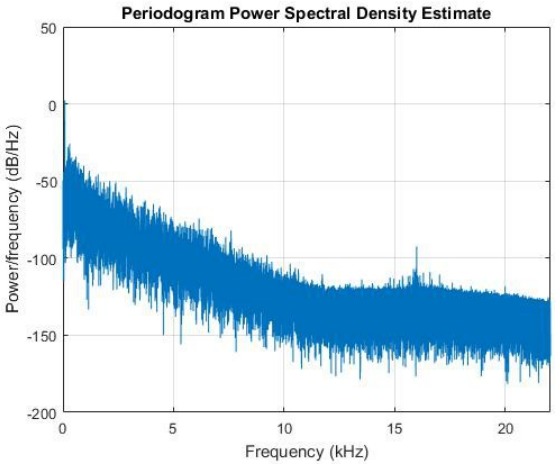


Average Power/Frequency = 2.72 * 10^(-3)

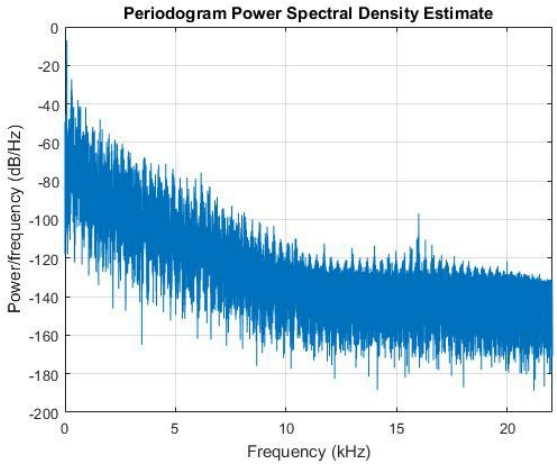


Average Power/Frequency = 0.93 * 10^(-3)

31. Sources on perpendicular axes, d=0.005m, piano and 300Hz

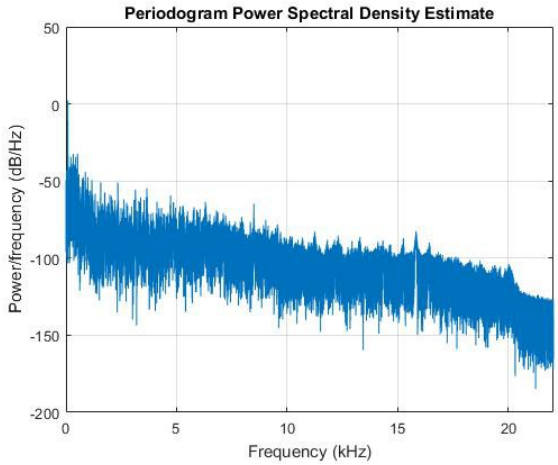


Average Power/Frequency = 26.03 * 10^(-3)

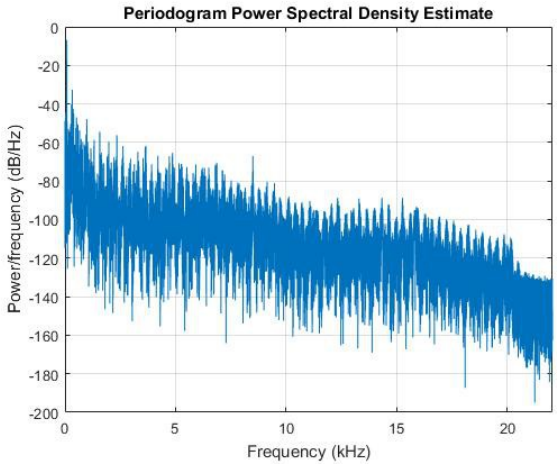


Average Power/Frequency = 4.01 * 10^(-3)

32. Sources on perpendicular axes, d=0.005m, guitar and 300Hz



Average Power/Frequency = 24.92 * 10^(-3)



Average Power/Frequency = 3.5 * 10^(-3)

Although the above presented results look very promising, as average power spectral density reduction is very significant, the author has doubts concerning the actual cause of this fact. For an empirical analysis, even though clearly using an array has some effect on perceived sound, its audible quality can hardly be described as spatial filtering. The resulting sound rather resembles a high-pass filter affected, in terms of frequency content. Some attenuation of one of sound sources can be observed, but it is very subtle. Probably, this is due to the fact that only three microphones were utilized and also the chosen basic method of delay-and-sum beamforming is known to exhibit rather underwhelming performance. Again, better results might have been achieved if the most explicit spatial configuration of sound sources was examined, with one being on the end-fire axis, and the other on the broadside.

7. Conclusion and future perspectives

With the implemented VST plugin and its underwhelming performance, the research question stated in the problem statement can sadly only be answered with a denial. It turns out that in order to achieve truly satisfying results, many more than three microphones must be incorporated. Furthermore, line array solution can only be used for very limited number of applications and even in this case, a sub-array technique would be more advisable. This, however, requires not only more sensors to be involved, but also a more sophisticated piece of software to handle the processing, dividing the range of interest into sub-bands.

The author believes that an approach taken by existing microphone array manufacturers, when a carefully designed hardware is coupled with powerful software is the appropriate one to tackle spatial filtering challenges. Still, novel techniques for array processing may lead one day to reducing the amount of channels involved and consequently making entry-level equipment and matching processors more accesible. So far, array processing is still a narrow domain, reserved for professionals aiming to minimize the amount of equipment required to record high quality, immersive audio.

With virtual reality becoming widespread these days, spatial audio is also becoming a trendy topic and therefore the author hopes that costliness of dedicated equipment will gradually decrease. Homegrown sound enthusiast will most likely play an important role in that process, but as it turns out from this project's findings, a delay-and-sum beamforming handling VST plugin does not suffice. Maybe if in future iterations more sophisticated approach would be taken, allowing better results with fewer sensors, shipped with readymade microphone capsules housing that would allow easy fiddling with a variety of possible array shapes, that would be more encouraging.

Bibliography

1. Simon Haykin, Array Signal Processing, Prentice-Hall 1985

2. Iain McCowan, Microphone Arrays: A Tutorial, 2001

3. F. Alton Everest, The Master Handbook of Acoustics, TAB Books, 1981

4. Hagit Shatkay, The Fourier Transform - A Primer, 1995

5. Berkhout et al, Acoustic Control by Wave Field Synthesis, 1993

6. P.B. Felgett, Ambisonic Reproduction of Directionality in Surround-Sound Systems, 1974

7. Harry L. Van Trees, Optimum Array Processing, Wiley 2002

8. Joerg Bitzer and K. Uwe Simmer, Superdirective Microphone Arrays, Microphone Arrays: Signal Processing Techniques and Applications, Springer 2001

9. Darren B. Ward, Rodney A. Kennedy, Robert C. Williamson, Constant Directivity Beamforming, Springer 2001

10. Simon Doclo and Marc Moonen, Superdirective Beamforming Robust Against Microphone Mismatch, 2006

11. Griffiths and Jim, An alternative approach to linearly constrained adaptive beamforming, 1982

12. Widrow and Hoff, Adaptive Switching Circuits, 1960

13. Simmer, Bitzer, Marro; Post-filtering Techniques, Springer 2001

14. http://www.tslproducts.com/soundfield-type/soundfield-microphones/

15. Fons Adriaensen, A Tetrahedral Microphone Processor For Ambisonic Recording, 2007

16. http://www.clearone.com/products_beamforming_mic_array

17. http://www.polycom.com/content/dam/polycom/common/documents/data-sheets/ceiling-microphone-array-ds-enus.pdf

18. http://www.core-sound.com/TetraMic/1.php

19. http://www.mhacoustics.com/products#eigenmike1

20. http://www.shure.com/americas/microflex-advance

21. http://www.idiap.ch/~mccowan/

22. http://www.dev-audio.com/products/microcone/

23. http://www.bksv.com/Products/analysis-software/acoustics/noise-source-identification/spherical-beamforming-8606?tab=descriptions

24. http://www.mathworks.com/products/phased-array/apps.html

25. http://www.vocal.com/beamforming/acoustic-beamforming/

26. http://www.vocal.com/resources/research/voice/#beamforming

27. https://sourceforge.net/projects/beamformit/

28. http://www.xavieranguera.com/

29. http://pcfarina.eng.unipr.it/Aurora_XP/Forthcoming%20modules.htm

30. http://pcfarina.eng.unipr.it/Aurora_XP/Package.htm

31. https://www.mhacoustics.com/sites/default/files/EigenUnits%20User%20Manual%20R01C.pdf

32. http://www.dlab.ch/arraybf.html

33. http://www2.st.com/content/st_com/en/products/embedded-software/audio-ics-software/open-audio/osxacousticbf.html

34. http://dspsoundware.com/demo-software/, https://community.arm.com/docs/DOC-9976

35. http://www.adaptivedigital.com/product/telephony/acoustic_bfmr.htm

36. Jacob Benesty, Jingdong Chen, Yiteng (Arden) Huang and Jacek Dmochowski, On Microphone-Array Beamforming From a MIMO Acoustic Signal Processing Perspective, 2007

37. Elmar Messner, Differential Microphone Arrays, 2013

38. Chao Pan, Jingdong Chen, and Jacob Benesty, Theoretical Analysis of Differential Microphone Array Beamforming and an Improved Solution, 2015

39. Hiroki Katahira, Nobutaka Ono, Shigeki Miyabe, Takeshi Yamada and Shoji Makino, Nonlinear speech enhancement by virtual increase of channels and maximum SNR beamformer, 2016

40. http://www.soundonsound.com/sos/1996_articles/jul96/steinbergcubase3.html

41. https://www.juce.com/case-studies/juce-cinder-in-conversation, https://en.wikipedia.org/wiki/JUCE

42. https://www.juce.com/doc/classes

43. https://courses.edx.org/courses/course-v1:Microsoft+DEV210x+1T2016/info

44. http://www.cprogramming.com/tutorial/c++-tutorial.html

45. https://www.juce.com/tutorials

46. http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html

47. http://www.audiomulch.com/

48. http://www.cstr.ed.ac.uk/projects/eustace/