Assignment 4: Eye-Gaze Steered Acoustic Beamformer

Introduction and Overview

Beamforming is a signal processing technique in which signals received from two or more sensors are processed together to emphasize and/or attenuate signals arriving from specific directions in space. Beamformers are useful for applications where one needs to focus on a 'target' signal arriving from a specific direction and ignore signals from other directions. One such application can be found when listening in 'cocktail party' settings through hearing devices (e.g. hearing aids and cochlear implants) that are worn by hearing impaired individuals who often have difficulties focusing attention on and understanding a specific target talker/music/sound source. However, for the user to be able to selectively 'steer' the focus of the beamformer towards a target signal, the algorithm would need to receive feedback from the user. One way of collecting feedback from the user is by measuring the visual electro-ocular data to estimate the user's eye-gaze angle, which could enable the user to steer the beamformer towards a target source by simply looking at it. Such an eye-gaze steered system could have other applications as well, such as optical exploration of acoustic surroundings either in real, augmented or virtual reality environments.

In this assignment, you will implement an acoustic beamformer that can be steered in real-time by the users' eye-gaze as estimated using electrooculography (EOG). The assignment consists of five parts. The first part involves background preparation that should be completed prior to the first lab session and that will cover material examined in the entrance test at the start of that session. The remaining four parts will be completed within lab and will guide you step-by-step through the implementation of the EOG steered beamforming system. The five parts of this assignment are:

0. Background Reading/Preparation: Read/watch the provided material listed below for background information on acoustics, beamforming, and EOG measurement
1. Introduction to audio processing: You will become familiar with playing, recording and processing audio signals in real-time. You will also be introduced to a real-time processing framework, which will be provided to you, that you will use in later sections
2. Beamforming: You will implement and test three simple beamforming algorithms using two microphones: delay-and-sum beamformer, differential microphone array and super-directive beamformer.
3. Real-time Eye-Gaze Estimation via EOG: You will implement a real-time eye-gaze tracker using EOG signals measured with the Biopac. You will be introduced to a number of challenges including measurement artefact removal, eye-movement classification, etc.
4. EOG steered beamformer: You will combine your real-time eye-gaze estimation system with your real-time beamformers.

0. Background Reading and Preparation

Prior to commencing the first lab session, you are required to read/watch the following materials to get familiar with the following topics. Note that there will be a marked entrance exam on the knowledge you gained during your preparation.

Fundamentals of Sound & Introduction to Real-Time Processing (online lecture):

A lecture series about 'Communication Acoustics', to which Prof Seeber of the AIP is a contributing lecturer, can be accessed free of charge on the online teaching platform EdX. Register to 'audit' the course (https://www.edx.org/course/communication-acoustics-rwthx-ca101#) and watch the videos in the following sections:

- 1.1 – 'What is sound?'
- 2.4 – 'Linear Time-Invariant Systems'
- 2.5 – 'Digital Filters'
- 2.6 – 'Digital Signal Processing Systems'

Acoustic Beamforming

For an introduction to beamforming, read Section 1.1 of Benesty and Chen (2013).

Electrooculography

Read through complimentary Biopac article titled 'Electroculogram I Introduction' and paper by Manabe et al. (2015).

1. Audio Signal Processing with Real-time Framework

This introductory part is supposed to make you familiar with the audio-hardware and -software which will be used for the remainder of the assignment. You will set up an audio device so that it can be used in real-time with MATLAB. Therefore, connect the audio interface M-AUDIO FAST TRACK C400 to a USB port of your computer. If working on a LINUX machine, use alsamixer to map the USB playback channels 1 and 2 to the headphone output by increasing the volume of the mixer channels labelled PCM1-OUT1 and PCM2-OUT2 (1st and 8th channel). With Windows this step is not needed. To access the audio device from MATLAB, we will use the Playrec utility (see http://playrec.co.uk/ for documentation and further information). Use one of the two precompiled functions (playrec.mexa64 for Linux or Playrec.mexw64 for Windows) and put it in a directory which is on your MATLAB path (if necessary, use addpath.m or the "Set Path" menu option to add a new directory to your path).

1.1. Playing Sound with Playrec

Playrec, contrary to MATLABs built-in audio tools like `sound.m` and `audioplayer.m`, requires the explicit selection of an audio device. In the "functions" folder of your downloaded files you can find a MATLAB function `playrec_init.m` that automatically identifies the M-AUDIO FAST TRACK C400 audio device and sets-up 2 recording channels and 4 playback channels at a sampling rate of

f_S = 44.1 kHz. By default, `playrec_init.m` sets the number of samples per buffer to 512, but accepts different buffer sizes as an optional parameter. Now use this function to set up your audio device.

Attention: ALWAYS protect your ears! Turn the headphone level knob (the small one above the headphone symbol) all the way to the left (i.e. down) now and adjust it gradually while playing back sound next and later.

Generate one second of white Gaussian noise with the Matlab function `randn`, apply a linear gain so that its amplitude never exceeds 1, and play it simultaneously to the left and right channels of your headphones using `playrec('play', playSignal, playChanList)`. The noise should be generated as a mono signal (i.e. an Mx1 column vector), which can then be converted to a two-channel signal (i.e. Mx2 vector) through appropriate matrix multiplication. For the most soundcard set-ups, setting `playSignal = [1,2]` will play to left and right headphone channels. Otherwise, the maximum number of available play channels can be determined using `playrec('getPlayMaxChannel')`.

If you want to set up a different device, you first need to call `playrec('reset')`.

1.2. Recording Sound with Playrec

Next, you will be made familiar with the recording functions of Playrec and use them to record sound samples that can be used for later tasks. The command `pageNumber = playrec('rec', recDuration, recChanList)` can be used to record sound, where the output variable pageNumber provides a pointer to the Playrec buffer to which the recorded audio data is stored. The stored data can then be retrieved using `recBuffer = playrec('getRec', pageNumber)`, which should be done AFTER the audio data has been recorded. To force MATLAB to wait until recording has finished, before retrieving the recorded data, use the command `playrec('block', pageNumber)`.

Connect one of the BEYERDYNAMIC TG V35ds microphones to an XLR mic input of your soundcard, and adjust the gain using input gain knob while speaking into the microphone. Each knob has a level meter that indicates how much of the devices' input dynamic range is being used. As the microphone and soundcard each introduce inherent noise to your recording, the input gain and your speaking level should be set to use as much of the input dynamic range as possible to achieve the best signal-to-noise ratio (SNR), without exceeding the upper limit of the devices' dynamic range (indicated by red bars on the input level meter) where undesired signal clipping can occur. Note that the mic has an on/off switch and that the '+48'V phantom power switch on the sound card should be turned off.

Record two speech samples of roughly 10s in length, each spoken by a different talker. Try to avoid long pauses when recording the speech. Plot each sample and verify that their amplitudes sufficiently occupy the recorded digital dynamic range (-1 to +1) without exhibiting clipping. Adjust the length of the samples and their (broadband) RMS levels to match one another, ensure that neither signal exceeds ±1. Save the two samples separately.

1.3. Playing and Recording Sound Simultaneously with Playrec

Sound can be simultaneously played and recorded using the command `pageNumber = playrec('playrec', playBuffer, playChanList, recDuration, recChanList)`, which will be necessary for later tasks. To help familiarize you with this command, we will use it to determine the delay (latency) introduced by the soundcard and USB link with your computer.

Generate a click signal in Matlab and play it from your headphones while simultaneously recording it by placing your microphone close the headphones. Estimate your system latency by comparing the recorded signal with the playBuffer input variable, first after initializing your audio device with a buffer size of 512 samples, and then with a buffer size of 2048 samples. Does the buffer size affect your system latency? Why?

1.4. Playing, Recording and Processing in Real-Time

The next step is to do real-time processing of audio signals. This can be achieved in MATLAB by continuously recording an ongoing input audio stream into buffers, applying the desired processing to each recorded buffer, queueing the processed buffers when ready and then playing them back at the correct time to reconstruct an ongoing output audio stream. To achieve a smooth uninterrupted output audio stream, a delay needs to be introduced between input and output audio streams, so as to give each record buffer time to be filled and processed before being played-back. Figure 1 illustrates the steps of a real-time processing system based on Playrec that runs in an ongoing loop. During the initialization phase before the real-time processing loop commences, two empty record buffers are setup and pointers ('page numbers') to those buffers are saved. During the loop, the program needs to execute the following steps: 1) wait for the next record buffer to be filled; 2) fetch the filled record buffer and process it; and 3) queue the processed buffer and play it back. Note that a call to `playrec('playrec',... )` returns immediately, which is a prerequisite for real-time processing. However, correct timing of step 1 of the loop (waiting until the next buffer can be read) can be achieved using `playrec('block', pageNumber)`, which pauses the program execution until the specified buffer is ready.

To facilitate the implementation of real-time processing algorithms, and to enable the interactive manipulation of processing parameters, you have been provided a simple framework in the function `realtime_processing.m`. The function takes the parameters `time, algo, initdata, initparam, playChanList and recChanList`. The parameter `algo` provides a handle to a function containing the processing algorithm (syntax: @<functionname>) that you wish to apply to each record buffer before playback. The processing function needs to be structured like the `dummy_algorithm.m` provided to you. All other input parameters are optional. `initdata` allows input of data required for initialization calculations that occur before the real-time loop begins and is passed to the 'init' phase of the processing algorithm. `initparam` enables input of the starting values used by the processing algorithm. Use of the two `init` inputs will become more evident in the next section. The last two parameters of `realtime_processing.m` are optional and allow you to choose the input and output channels used.

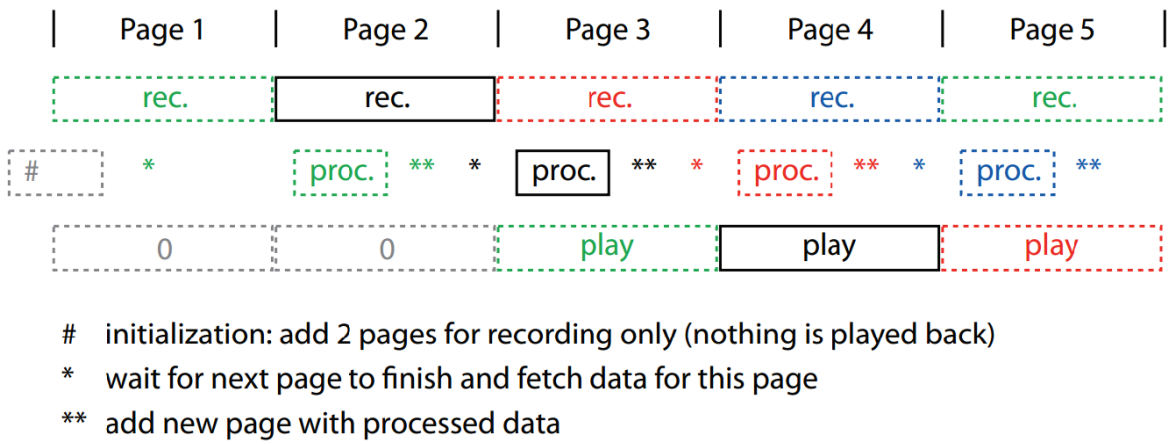

Figure 1: The processing steps of a real-time processing system using playrec. In the initialization stage (#), two audio buffers (referred to as 'pages' is the playrec documentation) are added for recording and their pointers ('page numbers') need to be stored to enable their retrieval later. In the real-time processing loop, the following steps need to be executed: 1) wait for the next buffer to be filled and ready; 2) fetch the recorded data from the next filled buffer, 3) process the retrieved data; queue them to be played. Note that a call to playrec('playrec',...) returns immediately, which is a prerequisite for real-time processing.

Task:

Modify `dummy_algorithm.m` to create a function `gain_ILD.m` that can be used to apply an overall gain specified in decibels simultaneously to both playback channels, and to apply a level difference between the two playback channels to produce a so-called 'interaural level difference' (ILD). Set the range of the adjustable overall gain parameter to be between -15 dB and +15 dB (the `'getparmranges'` phase) and the range for the ILD to between -3dB to 3dB, where a negative ILD corresponds to higher level at the left headphone channel (and vice versa). Make sure you also provide names for each parameter by adjusting the output of the (`'getparamnames'`). Run the `realtime_processing.m` and try modifying the playback gain and ILD in real-time using the GUI provided. Note: only the time and `algo` input variables need to be provided in this exercise; `initdata` and `initparam` can both be set to zero.

Do you perceive any changes while listening to playback over headphones while adjusting the ILD parameter?

1.5. Applying Relative Channel Delays in Real-Time

When delaying the output of one channel relative to the other, two factors need to be considered. Firstly, any delay applied to a channel must be CAUSAL (otherwise your system would have to look into the future!). Secondly, when processing block-wise, save buffers are needed to enable access to samples from previous frames to ensure smooth delay and playback.

Task:

Write a function `gain_ILD_ITD.m` that builds upon `gain_ILD.m` and that can be additionally used to delay one playback channel relative to the other to apply a so-called 'interaural time difference' (ITD). Set the range of adjustable ITD values to be between -1 ms and 1 ms; a negative ITD should

correspond to a leading signal at the left headphone channel (and vice versa). Remember to add the name for ITD parameter in the `'getparamnames'` phase of the function.

How does adjusting the ITD parameter affect the perceived sound over headphones? How does the perception change when both ILD and ITD are adjusted?

2. Microphone Arrays and Beamforming

Microphone arrays comprise two or more microphones that are positioned at fixed distances from one another to simultaneously record a sound field at different points in space. They are used in a number of applications such as acoustically determining the direction of a sound source and for blind-source separation algorithms that attempt to extract a specific signal from mixture of multiple concurrent signals. Microphone arrays can also be used for beamforming, whereby the signals recorded by individual microphones are combined to emphasise/cancel-out signals arriving from a specific direction. In this section, we will examine and implement three basic beamforming algorithms using a 2-microphone array.

2.1. Microphone Array Equalization + Loudspeaker Equalization

A frequently occurring problem in sound processing is that even recording and/or playback devices fabricated by the same manufacturer often exhibit sound characteristics that deviate from one another with regards to, for example, (frequency) magnitude and phase response, overall output level, etc. Therefore when using arrays of microphones and/or speakers, the devices within each array need to be calibrated. Measurement microphones are calibrated with a calibration device that outputs sound of a known level directly into the microphone to enable calculation of the scaling factor between physical acoustic level (e.g. in dB SPL) and digital level output by the microphone (between -1 and 1). Conversely, loudspeakers are typically calibrated by playing a stimulus of known voltage and spectral content from each loudspeaker and determining the physical acoustic level output with a sound level meter or measurement microphone positioned at the intended listening/receiving position, so as to compute a scaling factor between the voltage level input to the loudspeakers (or digital level if DA-converter, amplifier and loudspeaker build a fixed amplification chain as it is here) and the physical level output at the listening/receiving position. For the purposes of this assignment, it is however sufficient to simply equalize both microphone and both speakers to the same overall (frequency independent) level without necessarily determining the scaling factors between physical and digital levels.

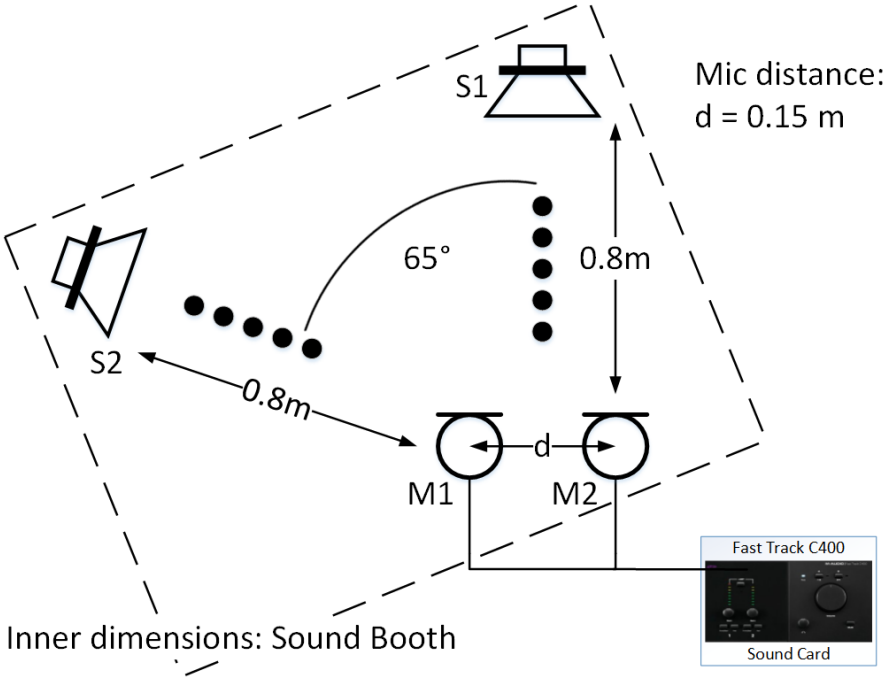

Figure 2: Configuration of the loudspeakers and microphones setup in the sound booth, with an inter-microphone distance of d = 0.15m.

Task

Using Playrec, write a script that calculates scaling factors for each microphone and each speaker such that the microphones output the same digital level to each other in response to the same acoustic stimulus, and the speakers output the same acoustic level to each other when playing the same stimulus. To save time, the loudspeakers and microphones have already been setup for you in the configuration shown in Figure 2.

- Generate a short white noise sequence and bandpass filter it to the approximate bandwidth of speech (100Hz to 8 kHz). As noise is random, compute the sequence once and re-use it for all calibration tasks.
- To calibrate the microphones to each other, play the calibration sequence from speaker S1 and record on both microphones simultaneously. Compute the RMS level of the signal recorded by each microphone, and compute scaling factors that equalize their RMS levels. What assumptions have been made for this equalization to be valid?
- To equalize the loudspeakers to one another, play the calibration sequence from each speaker separately (one after the other) and record using microphone M1. Compute the RMS level of the recordings for each loudspeaker and compute a scaling factor to equalize them in level. What assumptions have been made for this equalization to be valid?
- Store scaling factors for the microphones and loudspeakers in separate 2 x 1 vectors named `mic_scale` and `speak_scale` respectively, and save in .mat file. Do not change microphone amplification from now on.

The .mat file will be loaded in during the initialization phase of the real-time framework in the following section, and the scaling factors automatically applied.

2.2. Sample Microphone-Array Recordings for Offline Processing

In this task, you will play target and interfering signals from loudspeakers S1 and S2 respectively and record the generated sound filed with the microphone array. The recording will be used in later sections.

Task:

Using Playrec, simultaneously play a target signal from loudspeaker S1 and an interferer from S2, and record on both microphones. Use one of the speech samples recorded in Task 1.2 as the target signal and 100 Hz - 8 kHz bandlimited noise (noise-on-speech interference) for the interferer. Save the signals from the two microphones in a .mat file.

2.3. Delay-and-Sum Beamformer

The architecture of a delay-and-sum beamformer (DSB) is illustrated in Figure 3 for a linear array comprising N equally-spaced microphones, with a spacing of *d* meters between adjacent microphones. With such architecture, microphone signals are summed together, after individual delays and gains are applied to each signal. The output of DSB will be maximized when microphone signals are in-phase with each other, and thus sum constructively, and will be reduced when microphone signals are out-of-phase with one another due to destructive summations.

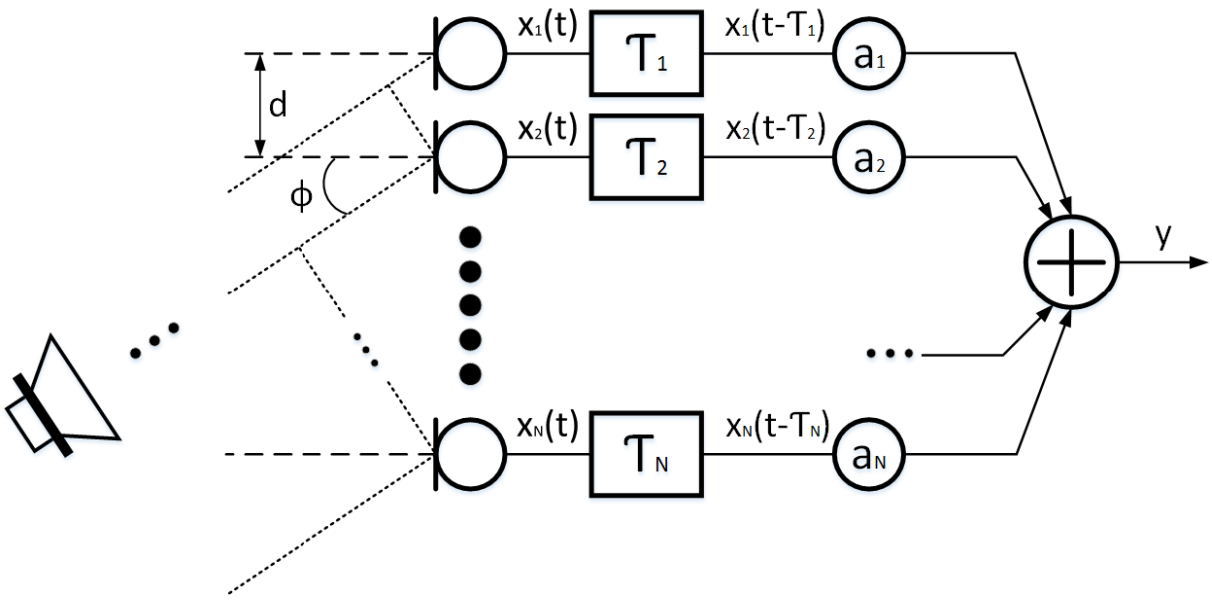

Figure 3: Outline of an N-channel delay-and-sum beamformer. Sound waves arrive at the microphone array as plane waves (assuming the distance of the source from the array is much greater than the distance between microphones) with different arrival delays. Delays are applied to each microphone channel to compensate for the arrival delays and realign the signals. The signals are then weighted and added together. Figure taken from Scheiner (2016).

Soundwaves originating from a source at angle ϕ will arrive at each microphone with different delays and thus with different phase shifts. When the sound source is sufficiently far from the array, the curvature of individual soundwaves impinging upon the array can be approximated as planes. The delay of the signal received by the n^{th} microphone, relative to microphone N, is therefore given by:

$$\tau_n = (N - n + 1)\frac{d}{c}\sin\phi$$

Equation 1

where c is the speed of sound (approximately 344ms^{-1} through air). From this equation, it can be seen that soundwaves originating from sources at angle ϕ = 0° (or 180°) will arrive at the same time at each microphone (τ_n = 0), yielding microphone signals that are in-phase. Sound from sources in other directions will instead result in out-of-phase microphone signals. Therefore, when no delays are applied to individual microphone signals (T_n = 0), the DSB output will exhibit a maximum output amplitude for soundwaves arriving from ϕ = 0°, as indicated by the peak ('main-lobe') in the gain vs source direction plot (Figure 6) of a 2-microphone DSB (blue curve). The main-lobe of the DSB output can be steered towards a desired angle φ by applying individual delays T_n to the microphone signal to compensate for physical delay at each microphone position as given by Equation 1, which bring the microphone signals in-phase for sounds arriving from the desired angle. The delays needed to achieve this are given by:

$$T_n = (n - 1)\frac{d}{c}\sin\varphi$$

Equation 2

The output of the DSB can be formally defined in the time and frequency domain as:

$$y(t,\varphi) = \sum_{n=1}^{N} a_n x_n(t - T_n)$$

Equation 3

$$Y(f,\varphi) = \sum_{n=1}^{N} a_n X_n(f)\, e^{-j2\pi f T_n}$$

Equation 4

It can be seen from Equation 4, the output of the DSB is frequency dependent, and when considered with Equation 2, this frequency dependency is affected by the steering angle φ and distance *d* between adjacent microphones in the array.

Task:

Adapt the `dummy_algorith.m` to write a function `DSB.m` that implements a real-time 2-microphone beamformer, where the steering angle can be adjusted with the slider in the real-time framework. For this, you may find the following suggestions useful

- The delays applied to each channel can be computed using Equation 2, but remember to convert those delays to samples rather seconds.

- Ensure that individual channel delays are always positive to ensure causality; negative delays will require your system to look into the future! Consider which of the two channels should be delayed, while the other is left unchanged, to ensure causality for a given steering angle.
- Since your function will process each incoming record buffer separately, you will need a save-buffer to store some samples from the previous record buffer, so that you can apply a delay in real-time. This save-buffer should be long enough to store enough samples for the longest expected delay.
- The steering angle should be accessible during runtime from the real-time frameworks slider. Define the range of that slider from -90° to +90° in the `'getparamranges'` section of your function, and assign the slider an appropriate name in the `'getparamnames'` part.
- To ensure the output amplitude doesn't increase relative to the input amplitudes, apply gains $a_n = 1/N$ to each channel, where N is the number of channels
- Play the output of the DSB to both headphone channels simultaneously
- As access to the speaker and microphone-array setup in the sound booth is limited, use the provided function `realtime_sample_processing.m`, rather than `realtime_processing.m`, to run your beamformer algorithm offline in simulated real-time. Supply the microphone array recording made Task 2.2 to the `realtime_sample_processing.m`.

Listen to the output of your DSB while adjusting the steering angle. Does your algorithm do a good job at focussing on sounds from a specific angle?

To examine the frequency characteristics of your DSB, filter your signals before they undergo the delaying and summing. This can be done by adding a filter command to the 'input' part of your function. The coefficients of the filter should be computed in the `'init'` part of your function, and saved in the 'state' structure. Separately try using a lowpass filter with cutoff f_c = 1.5 kHz, a highpass filter with f_c = 4 kHz, and a bandpass filter with f_c = [1.5 kHz, 4 kHz].

Listen to the outcomes. How does the apparent directivity of the DSB change when applying the different filters, and how does that compare to the unfiltered case? Can you think of a reason for differences?

2.4. Differential Microphone Array: Null-Steered Beamformer

Another simple beamforming algorithm known as a differential microphone array (DMA) is depicted in Figure 4 for a two-microphone array. As with the DSB, each microphone signal is delayed to compensate for the wave propagation delay between microphones associated with a desired steering angle. In contrast, the delayed microphone signals are subtracted from one another, cancelling or nulling sounds from the steering angle; for this reason, DMAs are often referred to as null-steering beamformers. The main advantage of a DMA is that is that it can effectively null sounds from the steering angle using a small number of microphones, as shown by the red curve in Figure 6 for a steering angle of 0°, though the width of the reject region can be narrowed by using more microphones. For our purpose, a drawback of a DMA is that it allows steering of null to suppress unwanted signals rather steering a beam to emphasis directions of interest.

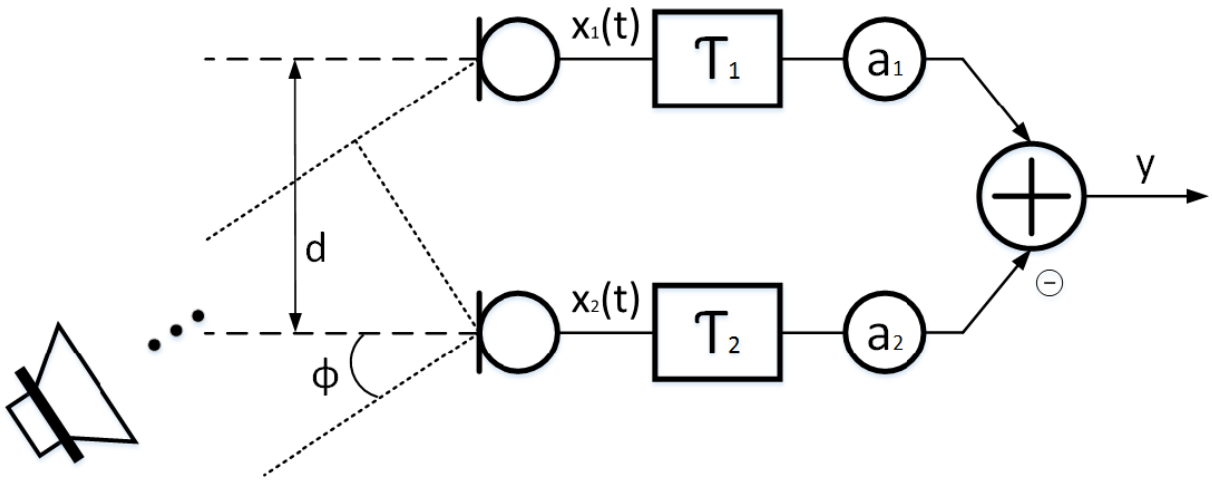

Figure 4: Schematic of a first-order differential microphone array (null-steering beamformer). Soundwaves arriving at the two microphones are weighted before one channel is subtracted from the other. Unlike the DSB, the steering angle is cancelled rather than emphasized. Figure taken from Scheiner (2016)

Task:

Adapt your `DSB.m` to write a function `DMA.m` to implement a real-time two-microphone DMA. Listen to the outcomes using the different filtered and unfiltered stimuli.

How does the outcome of the DMA compare to the DSB? Are there any apparent effects of signal frequency?

2.5. Generalized Side-Lobe Canceller

A third type of beamformer that we will examine is the so-called generalized side-lobe canceller (GSLC) that comprises a delay-and-sum that is used to emphasize sounds from the steering angle, plus a cancellation path that attempts to cancel-out sound from other directions. The architecture of an N-microphone GSLC is illustrated in Figure 5. As the response of a GSLC is highly frequency dependent, all processing is typically performed within frequency bands. B is an (N-1) x N matrix that is used to combine the delayed microphone signals to yield a DMA; when N = 2, B is set to [1 -1], to yield a 2-microphone DMA as in section 2.4. The null of the DMA is steered towards the main-lobe of the delay-and-sum path, since both paths receive the same delayed microphone signals. Therefore subtracting the output of the DMA will effectively cancel all directions except for the steering angle. The cancellation path also employs adaptive channel filters v_n, that are designed to maximize the power removed in the cancellation path. Those filters may be pre-computed and fixed, or continuously optimized at run-time. For more information of GSLC, refer to Griffiths and Jim (1982).

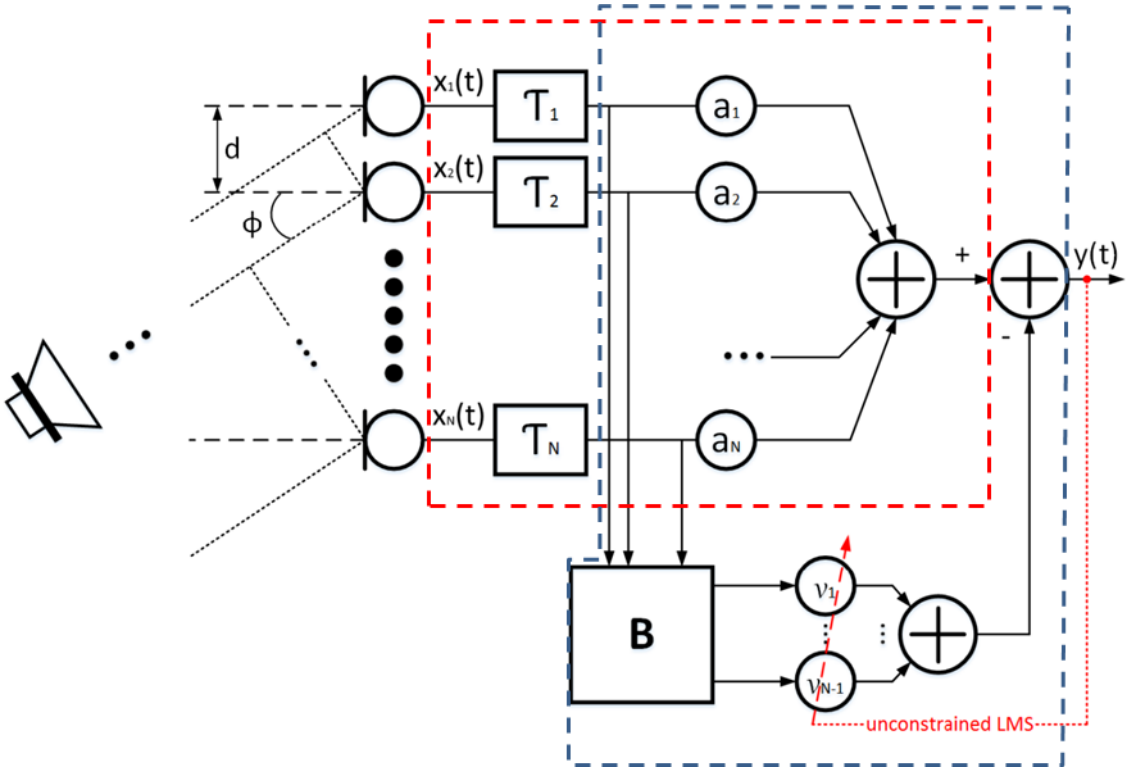

Figure 5: Block diagram of an N-channel generalized side-lobe canceller. It comprises a delay-and-sum part (enclosed by red box) and a cancellation path (section outside red box). The delay-and-sum part focuses on the spatial location dictated by the delays T_i. The cancellation part attempts to null that same direction using blocking matrix B, removing the signal of interest while leaving the interfering signals. Filters v_k are applied to each channel, and are optimized to maximize the output power of the cancellation part in a noise field. This signal from the cancellation part is then subtracted from the output of the delay-and-sum part. The processing steps enclosed by the blue box have been implemented for you in the function `GSLC_process_2ChanDelayed`

Task:

For this task, we will examine a two-microphone GSLC in which the filters v_m are precomputed and fixed to maximize power in the cancellation path in response to a diffuse noise field. As implementation of a GSLC is beyond the scope of this course, we have provided it to you a function `GSLC_process_2ChanDelayed.m` that implements the processing steps enclosed in blue box in Figure 5; i.e. the function takes in the delayed channels and generates the beamformer output. Refer to the function's help information for more details. Implement the provided GSLC beamformer in the simulated real-time framework and listen to its output for the different filtered stimuli.

How does GSLC compare to the other beamformers implemented so far? Are there clear frequency effects?

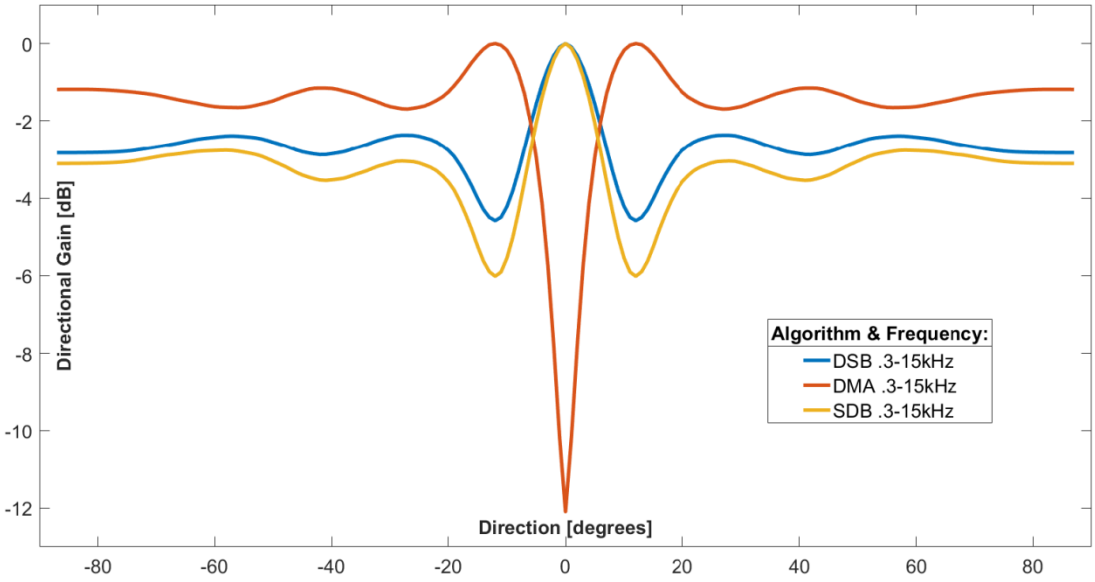

Figure 6: Comparison of the direction gains of the three beamforming algorithms in Sections 2.3 to 2.5, when using white Gaussian noise that has been bandlimited to the frequency range of 300 Hz to 15 kHz.

3. EOG

This task aims to develop a real-time system that estimates the horizontal angle of your eye-gaze, which will ultimately be used to steer the main-lobe of an acoustic beamformer.

Visual fixation is a process keeping an image of either a stationary or moving object projected onto a small area on the retina, known as the fovea, through fine muscular control. There are two types of visual fixation. To consciously fixate on a selected object in the visual field, abrupt voluntary eye movements known as saccades, which typically last 20-200ms, are used to move the gaze towards the object. If the object is moving, slower voluntary movements known as "smooth pursuit" are employed to track it. If the object is stationary, tiny involuntary unperceivable movements known as microsaccades are employed to counteract small relative movements between the object and eyes, and reduce perceptual fading caused by neural adaption to constant stimuli. Both voluntary and involuntary movements are controlled by six extraocular muscles.

Eye tracking and eye-gaze recording are important for numerous fields of research in medicine and neuroscience, as well as in psychology, market research and computer science (activity detection). There exist a number of approaches and implementations for tracking eye-gaze, which are summarized in Eggert (2007). Developments in optically based tracking systems (infrared or video-based) have yielded relatively small (even wearable devices) and accurate systems have gained much attention, but require huge computational cost when estimating gaze angles. Another widely used technique to track eye-gaze and movements is electrooculography (EOG), the processing of which demands lower computation power than the optical-based techniques. Further, EOG does not need a camera pointing at the eye, so vision, hearing and generally the head is unobstructed. EOG techniques have been implemented in portable devices (e.g. EOG goggles; Bulling et al., 2009), and could conceivably be embedded on mobile hearing devices, such as hearing aids or cochlear implants, to control the steering angle of an acoustic beamformer to help those users hear-out a target sound source in noisy situations.

EOG measures the direction of electrical dipole between the cornea (front) and retina (back) of the eyes, as depicted in Figure 7, using a set of electrodes (VIN+, VIN- and GND) placed in the configuration shown in Figure 8. Electrodes marked VIN+, VIN- and GND are used to detect horizontal movements. The unlabelled electrodes in Figure 8: Electrode place for measuring eye-movements with those needed for horizontal movements labelled Vin_+, Vin_- and GND. Those without a label are required to measure vertical movements and should NOT be attached. Figure taken from Figure 8 are necessary for detecting vertical movements and will not be used in this course. By measuring the potential difference recorded between VIN+ and VIN-, and assuming the eyes move in tandem, potentials arising from eye-muscle contractions and relaxations tend to cancel-out over the two eyes, leaving only potentials related to eye-dipole orientation. When the eyes are oriented to the right (see Figure 7), the positive pole of the right eye will be oriented towards electrode VIN+ while the negative pole of the left eye will be oriented towards VIN-, resulting in a positive potential difference between the two electrodes. Likewise for eye orientations towards the left, the positive pole of the left eye and the negative pole of the right eye will oriented towards VIN- and VIN+ respectively, yielding a negative potential between electrodes. When the eyes are instead facing straight ahead, the dipoles of the two eyes are perpendicular to the measurement direction, resulting in zero potential difference between electrodes. The potentials typically measured with EOG are on the order of a few micro-volts. As a result, EOG-based gaze-angle estimation typically has a resolution of around 0.5° due to signal-to-noise considerations.

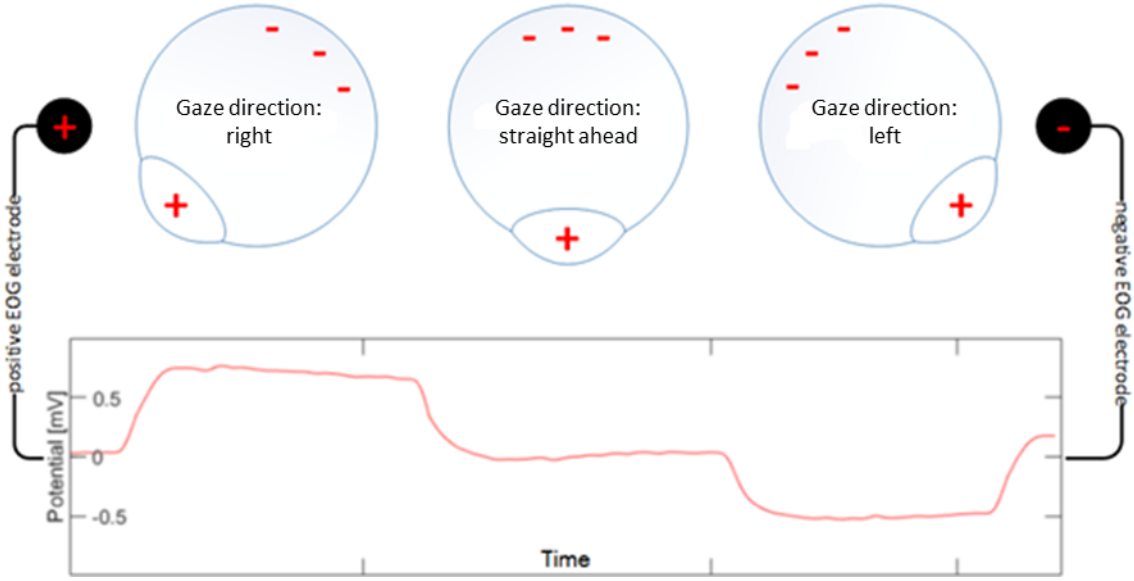

Figure 7: Depiction of the eyeball as a dipole orientated in different angles (top) and a resulting EOG signal recording between the electrodes Vin_+ and Vin_-, when the eye is fixated for short times. Figure taken from Fig. 4.4 of Scheiner (2016), and was redrawn from BIOPAC manual.

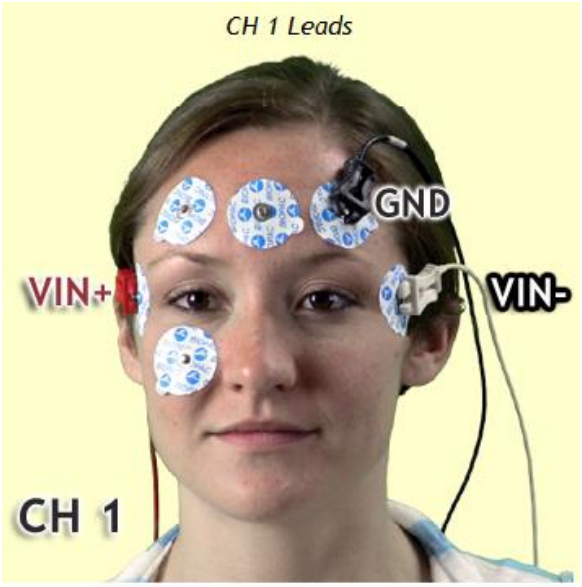

Figure 8: Electrode place for measuring eye-movements with those needed for horizontal movements labelled Vin_+, Vin_- and GND. Those without a label are required to measure vertical movements and should NOT be attached. Figure taken from Biopac manual.

The tasks in this section will first make you familiar with measuring EOG signals using the Biopac system. You will then develop your own EOG measurement routine that will be used to estimate the horizontal gaze-angle of the viewer in real-time.

3.1. Getting Started Measuring EOGs with Biopac

To process the potentials obtained with the electrodes, the signals have to be first amplified and digitized. This will be accomplished using the Biopac MP36 unit that allows simultaneous biosignal data acquisition on four channels. Like all mains-operated biosignal acquisition devices, the MP36 satisfies safety requirements specified in the 'Medizinproduktegesetz (German medical products law). For this reason the periphery of the Biopac system operates on a galvanically isolated safety extra-low voltage (SELV), which is safe for human and animal use.

Attention: To ensure your own safety, when performing measurement operations always make sure to not have any other equipment connected to the MP36 unit's input or output ports except electrodes and the USB connection to the computer!

Task:

Open the Biopac Student Lab (BSL; icon should be available on the desktop) and start the BSL lesson 'L10 - Electrooculogram (EOG)'. Follow the instructions of the experiment with one exception: as vertical eye movements are not our concern in this assignment only attach the three electrodes responsible for horizontal movement and connect them to channel 1 as described in the BSL (see also Figure 8). Do not connect the second channel and click on "Ignore" for corresponding

warning messages. Consequently you may skip both pencil-experiments examining vertical movements. Once you have finished the experiment, please leave the electrodes attached for use in the next task. You can remove the cables for the time in-between.

How do the EOG recordings differ when tracking a real versus an imaginary pendulum? From where do these differences arise? How do your eyes move when reading a text?

3.2. EOG Measurement in Real-Time

While the BSL software used in the previous task is a convenient tool to measure EOG signals and visualize them, it does not give you direct access to the signals in real-time. For that you will develop your own software in Matlab using some functions provided to you to interface with the Biopac MP36 unit. The script `biopacAPI.m` handles all communication tasks with Biopic. A short description of the functions contained within the script can found in MATLAB using '`help biopacAPI`' while more detailed information can be accessed with '`biopacAPI(0, 'help', command)`'.

The input command `'receiveMPData'` will instruct the biopacAPI to retrieve the next queued EOG acquisition buffer. There are two important differences to note between this command and Playrec (used for the audio real-time framework). First, the biopacAPI does not have a 'block' type command to pause your program until the next EOG acquisition buffer has been filled, which means you need to explicitly control when your program retrieves buffers. Second, the biopacAPI does not provide pointers to individual buffers in the acquisition queue, and only allows access to the first un-retrieved buffer in the queue at a given time.

You have also been provided the scripts `EOG_RT_Framework.m` and `RT_EOG.m`. The script `EOG_RT_Framework.m` provides a real-time framework, similar conceptually to the audio real-time framework introduced in Section 1, which executes an ongoing loop that can be used to retrieve and process EOG acquisition buffers in real-time. The script also contains an optional plotting routine that, when activated, can be used to visualize the received raw or processed EOG signals in real-time. The plot is updated at a lower rate than the EOG acquisition buffers are retrieved, since the plotting functions in Matlab can be computationally expensive. You won't need to make many changes to this script, except for selecting certain parameters such as the EOG acquisition runtime, how long the programs waits within the loop to retrieve an EOG acquisition buffer from the MP36, and parameters relating to the plotting function. These parameters have been marked with comments in the script.

Called within the real-time EOG loop is the `RT_EOG` function that you will be adapting throughout this section. The function is structured in a similar way to the `dummy_algorithm.m` introduced in Section 1, and comprises an initialization (`'init'`) and processing (`'process'`) phase. As before, the initialization phase is called once before the real-time loop in `EOG_RT_Framework` begins, and should be used to initialize variables and compute parameters that need to be computed only once upfront. The processing phase is continuously called within the real-time loop. The input/output structure '`state`' should be used to store any parameters that are needed to be carried over to subsequent calls of `RT_EOG`. The output structure `'output'` can be used to store a variables or signals that you want accessible after `RT_EOG` has returned.

Task:

Edit the function `RT_EOG` that will be continuously called by `EOG_RT_Framework.m` in real-time to retrieve EOG data from the MP36 acquisition buffers for a specified amount of time.

- In the marked section of `RT_Framework.m` outside the real-time loop, set the sampling rate of the MP36 (e.g. `state.fs` = 500; note that sampling rates used by the BIOPAC are considerably lower than those used for audio sampling) and the acquisition buffer length (e.g. `state.Nacq_buff` = 6). Set the '`plot_flag`' to 1 to activate the plotting routine and select how often the plotting routine should update. Set the run time of the program.
- As mentioned above, the `RT_EOG` should be only called to retrieve an EOG acquisition buffer when it is filled. Within the marked section of the `RT_Framework.m`, define '`wait_time`' to control how long the real-time loop should wait between subsequent calls of the `RT_EOG`. This time should take into consideration how long an acquisition buffer is expected to take to fill-up (which occurs in the background independently of your MATLAB scripts) and how long it takes for the commands to execute during each iteration of the real-time loop. Hint: the functions '`tic`' and '`toc`' can be used to measure time intervals in MATLAB.
- The command `biopacAPI(xxx, 'process',….)` should be called within the '`process`' phase of the RT_EOG function.
- You can generally add any fields you like to the `state` and `output` structures throughout the tasks in this entire section. However, a number of essential fields have been listed in the comments at the top of RT_EOG that are needed for the whole real-time framework to run properly. For example, `output.V_raw` should be used to output the raw EOG signal retrieved from acquisition buffer, so that the plotting routine in the real-time frame operates properly.

Once your program is up and running, use it to record an EOG data sequence for each of the three visual tasks described in the appendix, and save those sequences in separate .mat files. These three sample sequences can be used to test your code in later sections offline, without the need to connect to the Biopac. It is therefore important that you record those samples properly.

After you have measured your calibrations, and have recorded the sample sequences in the previous section, you can remove electrodes for comfort as you won't need them for the rest of this section.

3.3. EOG Filtering

Raw EOG signals typically suffer unwanted high frequency noise and low frequency baseline drift, both of which you may have already noticed in your EOG sample recordings. The high frequency noise can stem from the measurement hardware, antenna behaviour of the measurement electrodes and cabling, and incomplete common mode rejection of other undesired biosignals such as muscle contractions, microsaccades, etc. The low frequency baseline drift can arise from varying skin resistances, ocular resting potentials and slow changes in the composition of the electrode over time.

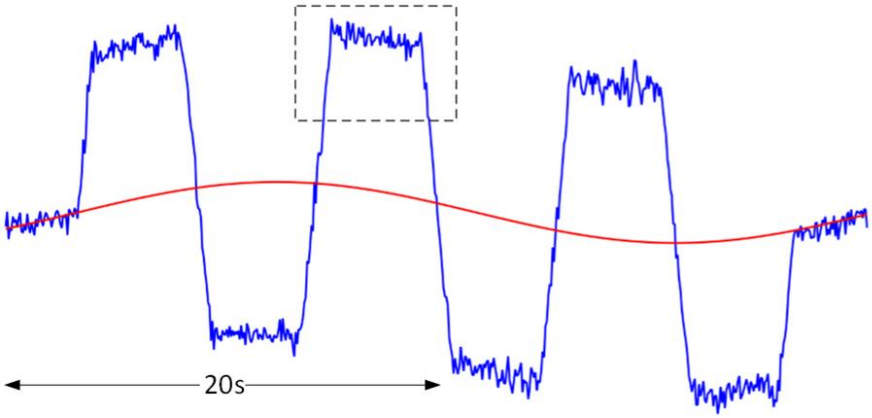

Figure 9: A measured EOG signal (blue) that is corrupted with high frequency noise (e.g. within broken box) and low frequency baseline drift (red)

Both artefacts are usually removed using a bandpass filter, often with the aim of retaining frequency content between 0.05Hz to 35Hz. An efficient implementation of this bandpass filter, proposed by Choudhury et al. (2005), is shown in Figure 10. It comprises a high-pass filtering stage in which a lowpass filtered version of the signal is subtracted from the original, and a standard lowpass filtering stage. The lowpass filtering stage employs a fourth-order type II Chebyshev filter (cheby2.m) while the high-pass stage uses a second order Butterworth (butter.m) lowpass filter.

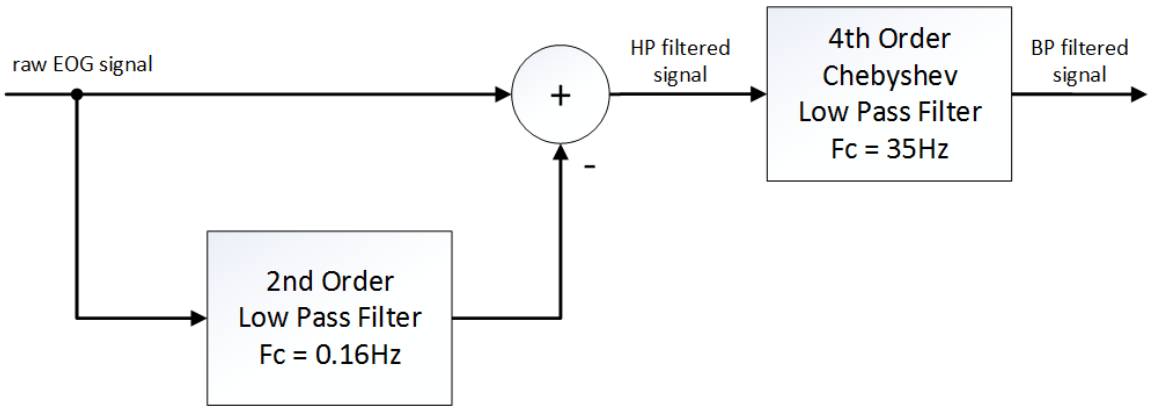

Figure 10: Implementation of a bandpass filter comprising a highpass stage followed by a lowpass stage to remove the low frequency baseline drift and high frequency noise. Figure taken from Scheiner (2016).

Task:

Implement and add the filters (`filter.m`) as described above to your `RT_EOG` function, incorporating the following suggestions:

- Use the `biopacAPI` in offline mode and load in one of your recorded EOG sample sequences.
- Make sure that you only compute the filter coefficients once during the initialization stage before the real-time loop commences.
- Remember to use the initial and final conditions of the filter delays, available with `filter.m`, to ensure smooth operation when applying filtering from acquisition buffer to buffer.

- For reasons that will become apparent later, **swap the order of the filtering stages relative to Figure 10,** so that the EOG data is lowpass filtered before being highpass filtered. This swap is valid due to the commutativity of property of linear filtering and will give you access to the lowpass and bandpass filtered versions of the signal.
- Output the lowpass and bandpass filtered signals via the output structure, with the field names 'V_LP' and V_BP' respectively.
- Save the plots at the end.

Our ultimate goal will be to steer a beamformer using the gaze-angle estimated from EOG signals, and have that beamformer remain orientated towards the direction that the eyes are fixated. When examining the bandpass filtered EOG signal, particularly when using your second EOG sample with infrequent eye-movements, can you identify any issues that would disrupt performance of the beamformer? What is the cause of this problem and why?

3.4. Saccade Edge Detection

As seen in Task 3.3, eye-gaze angles cannot be directly determined from either of the filtered EOG signals due to baseline drift in the lowpass-filtered signal and the issues associated with the bandpass signal. A common solution is to instead detect each eye-movement from the lowpass-filtered signal, determine the change in potential associated with that movement and then add that potential change to an ongoing cumulative estimate of a de-noised and driftless EOG signal. Of the two types of voluntary eye movements (see introductory part for Section 3), smooth pursuit movements can often be too slow to distinguish from artefacts in the EOG signal. We will therefore focus only on detecting saccadic movements. This will be achieved by identifying instances when the absolute derivative of the EOG signal crosses a defined derivative threshold, where the positive and negative-going threshold crossings represent the start and end of a saccadic movement respectively.

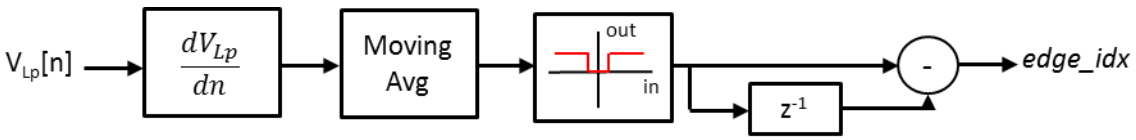

Figure 11: Block diagram of saccade detection algorithm. The derivative of the lowpass-filtered EOG signal ($V_{Lp}[n]$) is computed and smoothed with a moving average filter. The smoothed derivative is passed through a thresholding function that converts samples with absolute value exceeding a predefined threshold to 1, while those with absolute values below the threshold are converted to 0. This binary sequence is then delayed and subtracted from itself, yielding a sequence that contains 1s and -1s whenever the absolute low-pass filtered derivative crosses the threshold in the positive (saccade start) and negative (saccade end) directions respectively, and 0s otherwise

The block diagram for a simple saccade detection algorithm is shown in Figure 11 that identifies starts and ends of saccades from instances when the absolute derivative of the EOG signal crosses a predefined threshold in the positive and negative directions respectively. First, the derivative (dV_{LP}/dn) of the lowpass filtered EOG signal ($V_{LP}[n]$) is computed and smoothed with a moving average filter. The smoothed derivative is then converted to a binary signal via a thresholding function where samples with an absolute value exceeding a threshold are converted to 1, while all other samples are converted to 0. The binary sequence is then delayed by one sample and

subtracted from itself, yielding a sequence (edge_idx) comprising +1 at samples when the smoothed derivative signal crosses the defined threshold in the positive direction (start of saccade), -1 when a negative-going crossing occurs (end of saccade), and 0 otherwise.

Task:

Implement the saccade detection routine of depicted in Figure 11 into the real-time EOG framework.

- For the derivative calculation, it is sufficient to compute dV_{LP}, rather than dV_{LP}/dn. Think about whether you will need to save any samples from a past call to RT_EOG to compute dV_{LP}
- The moving average filter can be implemented as an FIR filter (try lengths around 10); don't forget to use the initial and final conditions for the filter to ensure smooth real-time operation.
- Try different values of derivative threshold (roughly on order of 0.001).
- Again, when performing the delay and subtract operation on the thresholded binary sequence to compute the edge_idx sequence, consider if you need access to samples of the binary sequence from previous calls of `RT_EOG`.
- Output the edge_idx sequence via the output structure

Try adjusting the length of your moving average filter and the level of the derivative threshold. What happens when the moving average filter length is too short? What happens if the derivative threshold is set too high or too low?

3.5. Estimating Changes in Potential during Saccadic Movements

Now that you have an algorithm for detecting when saccades start and stop, the next step is to determine the change in ocular voltage potential that results from those saccades, and sum those changes over time to generate a real-time estimate of the artefact-free EOG signal. This can be achieved by determining the potential directly prior to the start (initial level) and directly following the end (final level) of each saccade, while assuming negligible baseline drift over the duration of the saccade. To improve robustness, these potential estimates should be computed by averaging over a number samples preceding/following the saccade, rather than using the value of a single sample. The time frame over which averaging is performed (T_{avg}) should be long enough to supress rapid noisy fluctuations in the EOG signal but also short enough to avoid long-term baseline drift effects. When estimating potential changes, three different types of voluntary saccades (illustrated in Figure 12) need to be considered, the end points of which will likely be detected by your saccade edge detection algorithm. Normal saccades (points 1 to 2 in Figure 12) involve a direct movement from one eye fixation position to the next, with sufficiently low fixation durations before and after the saccade to compute initial and final potentials. Sloppy saccades (segment between points 4 to 7) involve an eye movement that comprises two or more smaller movements interrupted by short pauses (points 5 to 6). For such saccades, only the potentials before the start (point 4) and after the end (point 7) of the total movement are of interest, while the detected end (point 5) and start (point 6) of the subsequent sub-movements should be ignored. Momentary pauses due to sloppy saccades can be identified by instances when a detected saccade starts less than T_{sloppy} after a detected saccade end. Finally, 'rapid succession' saccades describe an instance when the eyes come to rest but do not remain fixated long enough to independently estimate the final potential reached by that saccade and the starting potential of the next saccade (points 7 and 8). Such a rapid movements may

be identified by an interval between the end of one saccade and the start of the next that is longer than T_{sloppy} but shorter than another defined interval T_{rapid}, where $T_{rapid} = T_{sloppy} + T_{avg}$.

For situations where rapid succession movements are absent, the initial potential should be estimated by averaging samples falling in the time frame T_{avg} directly before the detected saccade start. The final potential should be computed by averaging over time frame of T_{avg} after waiting T_{sloppy} relative to the detected saccade end to ensure that the eye has reached the intended fixation angle rather than being momentarily stationary due to a sloppy saccade, and to avoid the effects of signal transients associated with the eye coming to rest. When a rapid succession saccade is detected, the final potential of the leading saccade and the initial potential of the trailing saccade should be computed simultaneously by averaging over a shortened timeframe, relative to T_{avg}, before the start of the trailing saccade.

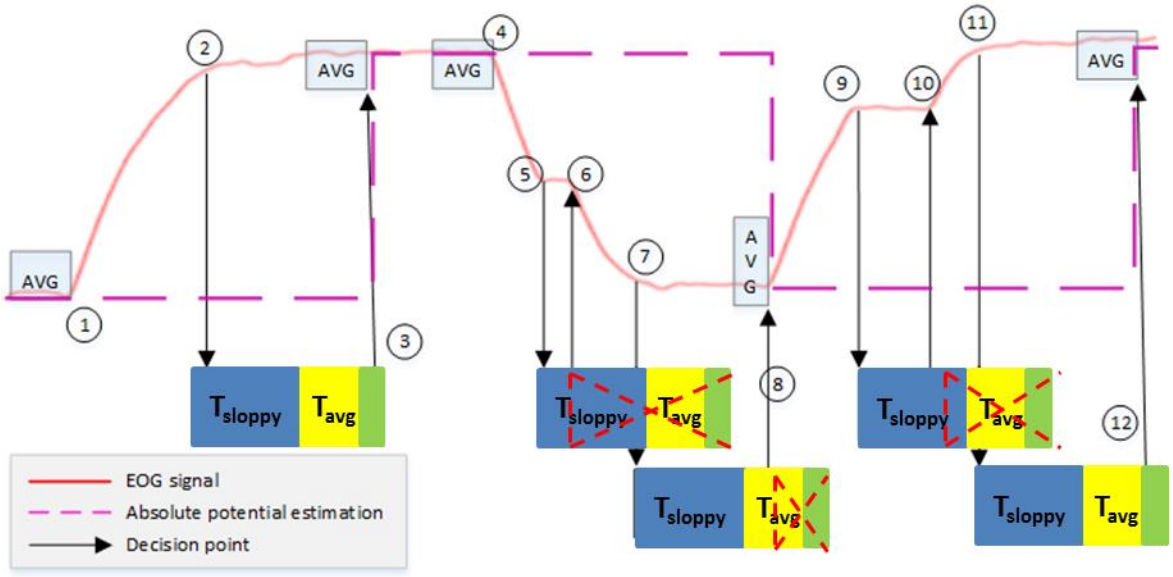

Figure 12: Estimating eye-gaze angle from an EOG signal while handling different types of saccadic movements. In this example EOG signal, an eye movement begins at (1) and ends at (2). The starting potential is computed within an averaging window at (1). A timer is initiated at (2). After the timer hits $T_{sloppy} + T_{avg}$, the movement is registered as a normal saccade, and the final potential at (3) of the EOG following the movement is determined by averaging over a window of nominal length (3). A new movement begins at (4), well after (3), and a new starting potential is measured with an averaging window. The movement ends at (5), which activates a timer. The time is halted at (6). Since this new movement occurs less than T_{sloppy} seconds after (5), a sloppy saccade is detected, and the movement end at (5) is ignored. A new movement end is detected at (7), which activates a timer. However, a new movement begins after the timer has exceeded T_{sloppy} but before $T_{sloppy}+T_{avg}$ is reached. A rapid movement is detected, and both the ending potential for (7) and the starting potential of (8) are computed together using a shortened averaging window near (8). The new movement undergoes a sloppy saccade from (9) to (10) and terminates at (11), which is registered as a normal saccade at (12). In all cases, whenever a saccade is detected, the overall potential estimate is updated by adding the potential difference between the end and the start of the saccade.

Task:

Add a routine to your real-time EOG framework that estimates the change in potential resulting from each saccade and add that potential change to an ongoing estimate of an artefact-free estimate EOG signal. A schematic of the cases that need to be handled by the routine is illustrated in Figure 12.

- The optimal selection of T_{avg}, T_{sloppy} and T_{rapid} can be tricky and may need to be adjusted ad-hoc. Previous evaluations revealed that roughly the following values yield acceptable performance: $T_{avg} \approx 100$ ms and $T_{sloppy} \approx 200$ ms.
- The averaging time frame, T_{avg}, is longer than the duration of a single EOG acquisition buffer. This means that you will need a way to access samples from previous acquisition buffers.
- To check if a detected movement end corresponds to actual end of a saccade, or a pause due to a sloppy saccade, a timer should be set-up from the instance when the movement end is detected. If a new movement is detected before the timer reaches T_{sloppy}, a pause due to a sloppy saccade is registered, and both movement edges bounding that pause are discarded. If the start of a new movement is not detected before time reaches T_{sloppy}, the detected movement edge is registered as the end of saccade, which triggers the potential averaging process.
- Once the averaging process is triggered, another timer should be activated. If the start of a new movement occurs before that timer reaches T_{avg}, indicating a rapid succession saccade, all samples collected since the averaging process commenced will be averaged to yield both the final potential of the previous saccade and the starting potential of the new saccade. If not, the ending potential of the saccade can be computed by averaging over all samples within the T_{avg} timeframe.
- Once both edge potentials of a saccade are computed, subtract the initial potential from the final potential to get the potential difference. Add that difference value to a cumulative estimate of the artefact-free EOG signal, which should be tracked using the state structure. Note that since cumulative EOG estimate should not change more than once while processing a single acquisition buffer, a single valued EOG estimate should be produced by each call to `RT_EOG`.
- For plotting purposes, also assign the scaler EOG estimate to `output.V_est`.

Update the plotting routine in the real-time EOG frame work to additionally plot your artefact-free EOG signal estimate, and compare it to the bandpass filtered EOG signal. How does your artefact-free estimate compare to the bandpass filtered EOG signal? What implications do the T values (described above) have on the responsiveness of the system? What effect does the outputting of a single value of the artefact-free estimate per function call have on the effective sampling rate?

3.6. Eye-Gaze Angle Estimation

By now, you should have a real-time EOG measurement system that outputs estimates of the EOG signal with artefacts like baseline drift removed. The next step is to convert those estimate EOG potentials to eye-gaze angles. This requires a mapping function that relates the potential differences recorded by the electrodes (V) to the gaze-angle of the user's eyes (θ). A linear relationship with a zero y-intercept can be assumed (Kumar and Poole, 2002) and can be fit by determining the potentials V_{calib+} and V_{calib-} that result from gazing at angles of $-\theta_{calib}$ and $+\theta_{calib}$ respectively. To

improve the robustness of the fit, θ_{calib} should be set to (or close to) the user's maximum gaze angle (40 to 50°), and V_{calib+} and V_{calib-} should be determined by averaging over multiple measurements.

Task:

First, compute the gradient of the EOG-potential to eye-gaze function and save it in `EOG_calib.mat`. V_{calib+} and V_{calib-} can be determined by first processing the EOG calibration sequence with your system and passing the artefact-free EOG estimate through the function `comp_Vcalib.m`. The artefact-free EOG estimate can be accessed via the variable '`EOG_Vest`' after the real-time EOG processing loop has finished.

Once you have determined the calibration gradient, use it to convert the artefact-free EOG potential estimates to eye-gaze angle estimates.

Note: Up-to-date calibration is essential for the conversion between EOG potential to gaze angle to be effective. Therefore it is advised to remeasure the calibration gradient frequently when operating online. When operating offline, compute the calibration gradient using the EOG calibration sequence recorded in Task 3.2.

3.7. System Recalibration

Despite your best efforts to eliminate baseline drift and other artefacts, you will most likely notice that your EOG/angle estimate signals will still drift away over time. Most EOG systems deal with this problem using some sort of re-calibration mechanism such as requiring the user the gaze at a defined angle (i.e. 0°) and press a reset button. Rather than pressing a reset button, we will attempt to cancel small drifts by purposely rounding down our cumulative angle estimation signal to 0° whenever the estimated angle falls between $-\theta_{min}$ and θ_{min}. Assuming that the user gazes towards 0° more frequently than the angle estimation signal can drift away by $\pm\theta_{min}$, the system should become re-calibrated. The drawback, however, is that small angles less than $\pm\theta_{min}$ around 0° will no longer be detectable.

Task:

Implement the above described recalibration mechanism whereby estimated angles that fall within the range of $\pm\theta_{min}$ are rounded down to 0°.

Can you think reasons why your gaze-angle estimation system would drift-off over time, despite efforts taken to eliminate baseline drift?

4. Eye-Gaze Steered Beamformer

Now that you have separately implemented a real-time acoustic beamformer and a real-time gaze-angle estimation system, it is time to put the two together!

The EOG (biopacAPI) and audio (Playrec) buffer acquisition processes can run independently on different processing threads in the background, but need to be retrieved and processed together in a coordinated fashion. The two data streams, however, operate at different sampling rates, with the audio stream (e.g. fs = 44.1 kHz) operating on the order of 100 times the rate of the EOG stream (e.g. fs = 500 Hz). Since the audio stream operates at a higher rate, the real-time retrieval and processing of the two data streams will be mediated by the audio real-time framework introduced in Section 1. The beamforming algorithms implemented in Section 2 will be adapted to additionally estimate eye-gaze angle using the `RT_EOG` function and use that estimate to steer the beamformer. Within the audio framework, one audio buffer should be retrieved on each iteration. However due to the lower sampling rate, EOG buffers should only be retrievable on a subset of iterations. Therefore, determining the iterations when the EOG buffer should be retrieved and processed is key.

In an ideal situation, the sampling rate and buffer sizes of the audio and EOG stream would set so that the time taken by an EOG buffer to fill would be an integer multiple N of the time taken by an audio buffer, which would allow an EOG buffer to be retrieved every Nth iteration. However, due to the possible sampling rates employed by Playrec and the Biopac system, an integer multiple is not possible. A counter is therefore needed within the audio framework to control on which iteration the RT_EOG function is called. An illustration of the buffering process is shown in Figure 13.

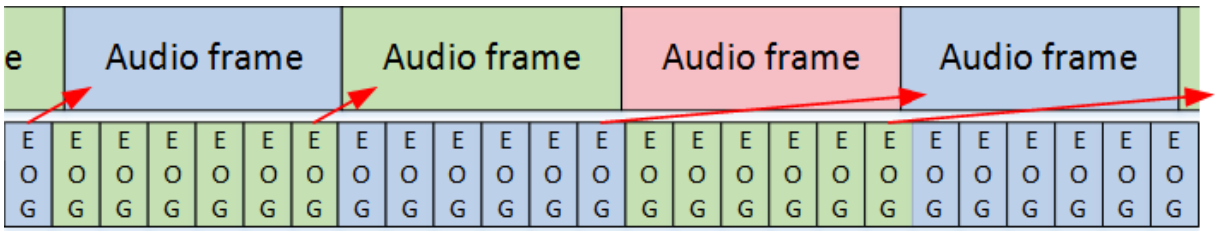

Figure 13: Illustration of the EOG buffer retrieval process while operating within the real-time audio framework. Successive audio frames (coloured rectangles), each of which comprise multiple audio samples are shown in the top. Individual EOG samples (vertical rectangular) are shown on the bottom, with samples sharing the same colour being collected by the same EOG record buffer. In this implementation, each EOG buffer can only be retrieved in the next audio frame that commences after the EOG record buffer is filled; red arrows indicate the audio frame at which each EOG buffer should be retrieved.

Tasks:

Adapt the beamformer algorithm developed in Section 2 that you feel yielded the best performance to incorporate your `RT_EOG` function to estimate the user's real-time gaze-angle and steer the beamformer main-lobe towards that angle. Run the eye-gazed steered beamformer within the offline audio-frame work.

- Execute the '`init`' and '`process`' commands of the `RT_EOG` function in the initialization and processing phases of your beamformer algorithm respectively.
- Make sure to not mix-up the state variables of the beamformer and `RT_EOG` functions.

- Implement a counter to trigger when the next EOG buffer should be retrieved, such that the buffer should be retrieved on the iteration AFTER it has been completely filled.
- If not done recently, remember to re-measure the EOG potential to gaze-angle parameters.

Test out your system and ensure that acoustic beamformer follows your gaze angle. How does the choice of sampling rate and buffer size employed by the audio and EOG streams affect performance of the system?

Appendix

Testing Procedures for EOG Experiment

Work in groups of a least two; one person acts as the test subject and the other the instructor. For all sequences described below, the test subject should look straight ahead at 0° before EOG recording begins. Once recording commences, the instructor should then tell the subject to look at specified directions at specified times (with the aid of a stop watch), as indicated in the tables below. Discuss where the subject should look before performing each sequence to avoid misunderstandings. Repeat this procedure for all three sequences.

Calibration Sequence

Before recording beings, the subject should fixate their gaze directly ahead at 0°. Once recording commences, they should alternate between gazing at -45° and +45° ten times, fixating at each angle for one second before switching.

Evaluation Sequence 1: Fast Angle Switching

The subject should transition from θ_a to θ_b, fixate at θ_b for 1 second and then transition back from θ_b to θ_a and fixate at θ_a for one second. This procedure should be repeated N number of times, before proceeding to fixate at a new set of angles. The fixation angle order is shown in Table 1.

Table 1: Fast angle switches every second for use when calibrating the EOG system.

Angles ($\theta_a \rightarrow \theta_b$ & $\theta_b \rightarrow \theta_a$)	**Repetitions per transition (N)**
0 to -45	1
-45 to 45 & 45 to -45	10 + 10
-30 to 30 & 30 to -30	5 + 5
-10 to 10 & 10 to -10	5 + 5
-5 to 5 & 5 to -5	5 + 5
-2 to 2 & 2 to -2	5 +5
-2 to 0 & 0 to 2 & 2 to 0 & 0 to -2	3 + 3 + 3 +3

Evaluation Sequence 2: Low Fixations

The subject should transition from θ_a to θ_b, fixate at θ_b for the indicated amount of time, and then transition to the next angle. The fixation angle order is shown in Table 2.

Table 2: Long fixations progressing from right to left (after moving right from the mid-point at the start). Used for the filter evaluation and beamforming.

Angles ($\theta \rightarrow \theta$)	**Fixation Time (s)**
0 to 45	10
45 to 25	10
25 to 10	10

10 to 0	10
0 to -10	10
-10 to -25	10
-25 to -45	10

Sequence 3: Debugging EOG Steered Beamformer

The subject should transition from from θ_a to θ_b, fixate at θ_b for the indicated amount of time. The fixation angle order is shown in Table 3.

Table 3: Sequence of long fixation times used to debug EOG steered beamformer.

Angles (θ→θ)	**Fixation Time (s)**
0 to 45	10
45 to -25	10
-25 to 0	10
0 to 45	10
45 to 25	10
25 to 0	10
0 to -25	10

References

BENESTY, J. & CHEN, J. 2013. *Study and Design of Differential Microphone Arrays*, Springer.

BULLING, A., ROGGEN, D. & TROSTER, G. 2009. Wearable EOG goggles: Seamless sensing and context-awareness in everyday environments. *Journal of Ambient Intelligence and Smart Environments,* 1**,** 157-171.

CHOUDHURY, S. R., VENKATARAMANAN, S., NEMADE, H. B. & SAHAMBI, J. 2005. Design and development of a novel EOG biopotential amplifier. *International Journal of Bioelectromagnetism,* 7**,** 271-274.

EGGERT, T. 2007. Eye movement recordings: methods. *Dev Opththalmol.,* 40**,** 15-34.

GRIFFITHS, L. J. & JIM, C. W. 1982. AN ALTERNATIVE APPROACH TO LINEARLY CONSTRAINED ADAPTIVE BEAMFORMING. *Ieee Transactions on Antennas and Propagation,* 30**,** 27-34.

KUMAR, D. & POOLE, E. 2002. Classification of EOG for human computer interface. *Second Joint Embs-Bmes Conference 2002, Vols 1-3, Conference Proceedings: Bioengineering - Integrative Methodologies, New Technologies.* New York: Ieee.

MANABE, H., FUKUMOTO, M. & YAGI, T. 2015. Direct Gaze Estimation Based on Nonlinearity of EOG. *Ieee Transactions on Biomedical Engineering,* 62**,** 1553-1562.

SCHEINER, N. 2016. *'Hear-out where you are looking' - 'Development of a Gaze-Steered Acoustic Beamformer.* Masters, Technical University of Munich.