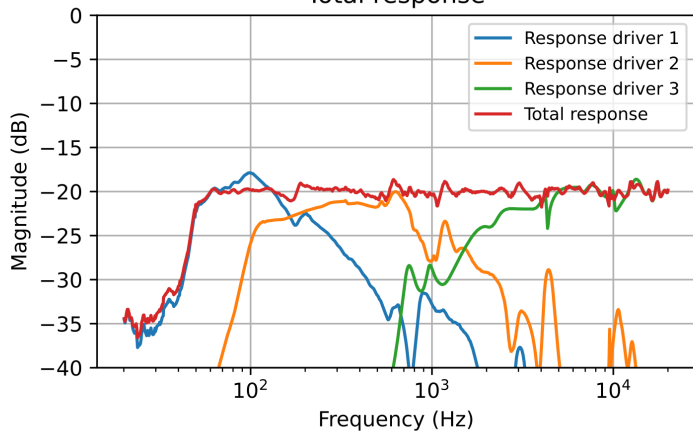


Loudspeaker Filter Design With AI

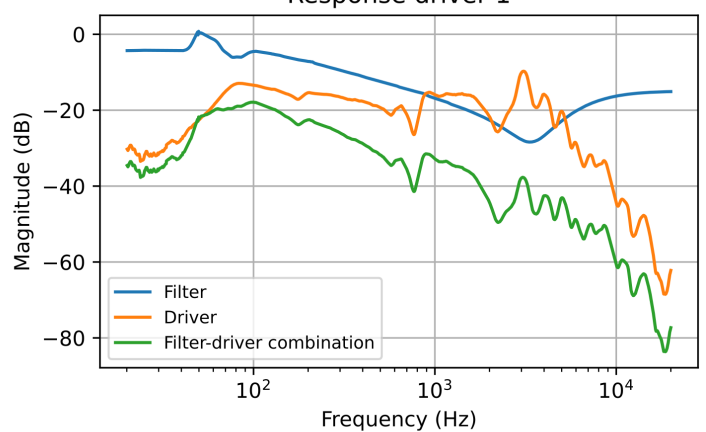
Communication, Evaluation and User Interaction

J.W. Nijenhuis & T.R. Zunderman

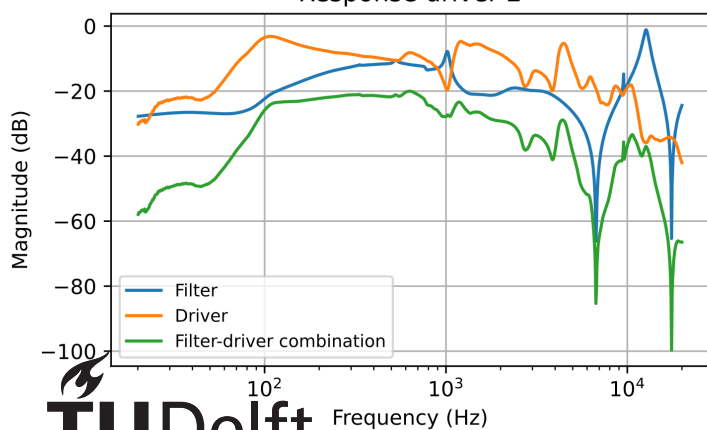
Total response



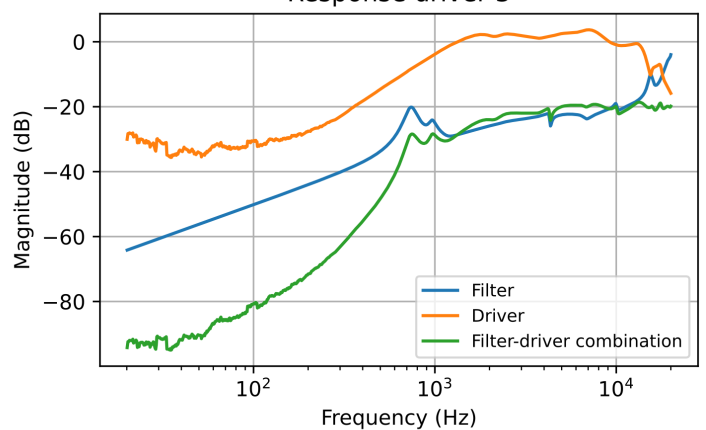
Response driver 1



Response driver 2



Response driver 3



Loudspeaker Filter Design With AI

Communication, Evaluation and User
Interaction

by

J.W. Nijenhuis & T.R. Zunderman

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Tuesday June 27, 2023 at 1:30 PM.

Student number:	J.W. Nijenhuis	5400503
	T.R. Zunderman	5293707
Project duration:	April 24, 2023 – June 30, 2023	
Thesis committee:	Dr.ir. G.J.M. Janssen,	TU Delft, supervisor
	Dr.ir. J.S.S.M Wong,	TU Delft, jury chair
	Dr.ing. R.K. Bishnoi,	TU Delft, jury member

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

This thesis describes the design and implementation of a controller and GUI for an AI loudspeaker filter design program. This program uses a genetic algorithm to create a combination of analog passive filters, one for each driver in a loudspeaker system. In the controller, two cost functions are designed to evaluate these filter combinations, and when the genetic algorithm has created its final filters, unnecessary components are removed and all component values are optimized. A GUI is created to allow easy user interaction.

For selecting a cost function, not enough data was obtained to make a deliberate decision based on performance. Therefore, this decision was based on theory and subordinate features. Nonetheless, the final program is able to create flat acoustic responses in a margin of 2 dB.

Preface

'Loudspeaker Filter Design With AI: Communication, Evaluation and User Interaction' is written for the Bachelor Graduation Project of Electrical Engineering, to finish this bachelor at the Technical University of Delft. The subject was proposed by dr.ir. G.J.M. Janssen, who is also our supervisor. During the 'EPO-1: Booming Bass' bachelor project earlier this academic year, a few students tried to use some form of AI to find a more flat acoustic response. Mr. Janssen, as one of the supervisors of that project, wanted to see whether it would be possible to use AI for designing filters for a loudspeaker system, which we tried for the past weeks.

We would like to thank Mr. Janssen for all his time and expertise. It was exciting to hear you talk passionately about loudspeakers and their characteristics. Thank you for all your patience with us. We would also like to thank dr.ir. J.H.G. Dauwels, for judging us during the green-light assessment and giving us feedback. Lastly, we would like to thank Koen and Jeroen, and Zoyla and Willem for doing this project with us. Without you, the results would not be what they are right now.

*J.W. Nijenhuis & T.R. Zunderman
Delft, June 2023*

Contents

Acronyms	1
Glossary	2
1 Introduction	3
1.1 Background information	3
1.2 State-of-the-art analysis	4
1.3 Design and implementation restrictions	5
1.4 Thesis synopsis.	6
2 Program of Requirements	7
2.1 Global requirements	7
2.1.1 Mandatory Requirements	7
2.1.2 Trade-off requirements	8
2.2 Controller requirements	8
2.2.1 Mandatory requirements	8
2.2.2 Trade-off requirements	9
2.3 GUI requirements.	9
2.3.1 Mandatory requirements	9
2.3.2 Trade-off requirements	9
3 Controller	10
3.1 Speaker data	10
3.1.1 Logarithmically spaced data	10
3.1.2 Operating ranges	11
3.2 Components	11
3.3 Construction of the filter response.	11
3.4 Cost function	12
3.4.1 Weighted sum	13
3.4.2 Separate costs	14
3.5 Main loop	17
3.6 Removing components.	18
3.7 Last value optimization.	19
4 GUI	21
4.1 Python library for GUIs	21
4.2 Loading files	21
4.3 Multithreading.	22
4.4 Showing results.	22
5 Implementation, Validation and Discussion	24
5.1 Implementation process	24
5.2 Cost function validation	24
5.3 Removing components validation	25
5.4 Last value optimization validation	27
5.5 Product validation	27
5.6 Discussion	27
6 Conclusions	28
6.1 Future work	28

A	Figures of test speaker	29
B	Computing time of the cost function	32
C	GUI Windows	33
C.1	Start window	33
C.2	Loading data	33
C.3	Running the program.	33
D	Best result found	41
	Bibliography	51

Acronyms

AI Artificial Intelligence.

Att Attenuation.

CPD Cost Per Driver.

CPS Cost Per Section.

CPU Central Processing Unit.

GA Genetic Algorithm.

GIL Global Interpreter Lock.

GPU Graphical Processing Unit.

GUI Graphical User Interface.

MSE Mean Squared Error.

PoR Program of Requirements.

SPICE Simulation Program with Integrated Circuit Emphasis.

ToR Trade-off Requirement.

Glossary

child In each generation, on every parents either reproduction, mutation or crossover is performed. After one of these actions is executed, the parent has become a child. A more in-depth description of the reproduction, mutation and crossover can be found in [1].

generation In each generation of the Genetic Algorithm, new parents are selected from the current children and these parents are mutated to become new children again. The number of generations is how often this cycle is executed.

parent During the selection in each generation, a set of circuits is selected. Each of these sets of selected circuits is called a parent. Before the set is selected, it is called a child, and after the selection a parent.

population size The population size is the amount of children that is generated during the mutation process.

selection size The amount of times that a tournament is held in tournament selection is the selection size.

T-section A T-section, also knows as a T-circuit, consists of three components that are placed in a T, or Y, configuration. The component type is randomly chosen and can either be a resistor, a capacitor or an inductor. When a T-section is initialized, the value of each component is also randomly chosen.

tournament selection The selection method that is used to select the children that will become parents is tournament selection. This means that a tournament is held with a certain amount of randomly selected children, the tournament size, where the child with the lowest cost becomes a parent. The selection size is the amount of tournaments that is held. More information on this selection method and the parameters can be found in [2].

tournament size The tournament size is the amount of children that is put in one tournament in tournament selection.

tree Each circuit that is designed by this program is represented by a tree data structure. A tree consists of nodes and leaves, where each node describes how its two branches are connected (either series, parallel or cascade) and each leave is a circuit. All the circuits are T-sections. A more extensive explanation can be found in [1] or [3].

Introduction

In a world where Artificial Intelligence (AI) is becoming more and more relevant and new applications are invented every day, AI has been used to come up with better solutions for problems where only minor improvements were made in the last few decades. In the field of designing analog filters, AI is able to come up with multiple solutions that fulfill requirements in less time than humans could. In this project, the task is to implement such an AI to design analog passive filters for a loudspeaker system.

Currently designing such filters is a time-intensive, skill-required task. While it is possible to design an analog filter that gives a speaker system a flat acoustic response, there is no structured method to do so, as will be described in the state-of-the-art analysis.

The goal for this Bachelor Graduation Project is to make a program using AI that designs a set of filters for a speaker system, one analog filter for each driver, that results in a flat acoustic response for the whole speaker system.

1.1. Background information

An ideal loudspeaker would produce every frequency equally loud, but in reality, this is never the case. The acoustic frequency response of a speaker is never completely flat due to physical limitations. Moreover, their physical properties limit loudspeakers to only work effectively in a certain range of frequencies [4], and this range never spans the complete range of audible frequencies (20 Hz to 20 kHz). To combat this, speaker cabinets are designed with multiple drivers inside. The simplest speaker cabinet consists of a driver for low frequencies, also known as a woofer, and a driver for high frequencies, known as a tweeter. A third driver, called a mid-range driver, could be included for the middle frequencies. These two-way and three-way loudspeakers are most common, but sometimes, for extra low frequencies, a sub-woofer is added. Additionally, multiple drivers for the same frequency range can be used. An example of a more extensive loudspeaker cabinet includes two woofers, a mid-range driver and a tweeter.

Although this solves the problem of limited operating ranges, speaker cabinets still have two problems. First, even inside the operating range of a driver, the acoustic response of a driver is only moderately flat. Second, at the boundary of two operating ranges, the corresponding two drivers will interfere. This is because a driver still produces sound outside its operating range when no filtering is done. In this case, its frequency response is even less flat, and/or less power is converted into sound. To solve these problems, filters are used. Especially the second problem can be solved with crossover networks that only send each driver frequencies inside its operating range. For these filters, it is important to make sure that when all driver responses are added, their response is actually flat. Due to overlap, different roll-off characteristics and phase differences, peaks and valleys could be added to the total response. Solving the first problem with filters is more difficult, and less common. Removing the imperfections of the acoustic response of a driver cannot be done by simply filtering out some frequency bands, it requires complex circuits to slightly attenuate some frequencies more than others. Mostly, these imperfections are minimized in the design phase of the speaker cabinet itself.

1.2. State-of-the-art analysis

In the past, analog circuits were used to obtain a desired frequency response. Designing such audio filters, however, requires extensive knowledge. These circuits are designed by experts in the field who use a combination of intuition and trial and error with certain filter modules, like high-pass and low-pass filters. This, however, is not accurate and it may take a lot of time to tune the frequency response in order to make it flat. In 1995, Assured and Nielson stated the following [5, p. 6]:

Analog circuit design is known to be a knowledge-intensive, multi-phase, iterative task, which usually stretches over a significant period of time and is performed by designers with a large portfolio of skills. It is therefore considered by many to be a form of art rather than a science.

Before the automation of filter design, the design of analog filters followed a standard procedure [6], [7]. It consists of three general steps:

1. Approximation
2. Realization
3. Study of imperfections

The approximation step is concerned with the generation of a transfer function that satisfies the desired specifications, such as the desired amplitude, phase and time-domain response. During the realization step, the defined transfer function is converted into a circuit that specifies the requirements of the previous step. In the realization step, ideal circuit elements are assumed. In practice however, the filter circuits are implemented by means of non-ideal components, which suffer from tolerances, parasitic elements and non-linearities. During the last step, the effects of non-idealities are studied.

The current situation with regard to filters has changed from analog solutions to digital solutions. Using digital filters over analog filters has some major advantages: the filters can easily be made by computers and can make the final response, even if the acoustic response of speakers is highly non-ideal (i.e. not flat), much flatter than analog circuits can do. However, there is a group of people, audio hobbyists, or audiophiles, who still use analog filters for audio applications, simply because they prefer the art of analog designs or the sound of analog components. Besides, the quality of analog audio filtering is better, since there is no sampling needed to filter the signal, as opposed to digital filters [8]. Next to the advantages of digital filters, analog filters or circuits also have advantages: they are cheap, relatively small and widely applicable. There are also fields in engineering where there is no other option than using analog circuits, for example for power decoupling, filtering to prevent anti-aliasing, and band extraction.

For the cases where analog filters still are needed, there are nowadays different computer algorithms, i.e. AI, that are used to find analog filter circuits and the values of the components. For these algorithms, the type of filter, and the boundary conditions, like the cut-off frequency and the pass-band frequency are given as an input and the algorithm gives a circuit with values back, which will satisfy the given requirements. However, none of these algorithms use the acoustic responses of drivers in a loudspeaker system to design filters to achieve an overall flat acoustic response. These algorithms could nonetheless be used to design such filters.

[9] sums up different techniques using AI to optimize analog (integrated) circuits. Neural networks and reinforced learning make use of machine learning, where neural networks use a large amount of data examples to train a model that can predict the best result. Reinforcement learning uses rewards and penalties to encourage finding a solution with the highest reward. The second group of techniques is optimization algorithms. The most occurring in this group are particle swarm [10], [11], simulated annealing and genetic algorithm (e.g. [12]–[14]). Particle swarm optimization and simulated annealing are used for optimizing values when the topology is fixed, while genetic algorithms can both be for optimizing values in known topologies [14], [15] and designing a topology and optimizing the values [16], [17]. When optimizing values for components, both optimizing for ideal values and discrete values is possible [15]. It should be noted that when ideal values are found, they should be rounded to existing analog components. Another option is to search for a discrete set of values.

To evaluate the performance of a generated circuit, SPICE is often used [18]–[22]. In [18], the simulation takes around 90 percent of the total computation time, the exact percentage depending on the complexity of the circuit. [19] shows that by letting the program calculate and evaluate the transfer

function, the percentage of simulation was brought down.

In the future, AI will become more and more advanced. In addition, analog filters will still be relevant despite the existence of digital solutions. For example, the output of an electrical transducer still needs to be filtered in order to effectively process it digitally. It is therefore expected that the tools to automate analog filter design will continue to improve. Analog filters will remain necessary.

1.3. Design and implementation restrictions

The AI will initially be developed for a three-way speaker since this requires a low-pass, band-pass, and high-pass filter. A two-way speaker has no mid-range driver, so it does not need a band-pass filter, and four-way and five-way speakers only need more band-pass filters. In other words: if the program can design filters for a three-way speaker, it is expected to be able to design filters for an N-way speaker with some minor changes. The component values will be chosen from a discrete set, since the final circuits should be physically realizable, without combining many components to create non-standard values. A method is chosen that will not require the use of SPICE. The main advantages of not using SPICE are that we can keep the whole program inside one environment and it will be beneficial for a lower runtime. The environment chosen to implement this is Python [23]. Python has some advantages, e.g. there are many libraries available that can be used for specific functionalities and Python makes use of classes, which makes developing structures of code more easy.

In order to achieve the required acoustic response, a genetic algorithm (GA) was selected to design the combination of analog circuits, as it is often used for this purpose in literature [12]–[14], [18]–[21].

When using a GA, an initial population is made. Then, using a cost function, the performance of each individual (child) is measured and a predefined amount of children is selected to go to the next generation, turning them into parents. These parents are then mutated or reproduced and become children, after which the children are selected and turned into parents until a child is formed that meets the predefined requirements.

In the case of a three-way speaker, the initial population consists of three circuits per child, one circuit for each driver. A total cost is calculated per child, taking the designed filters in combination with the loudspeaker system into account. To be able to estimate the performance of each set of filters, the acoustic response (magnitude and phase) and the impedance (magnitude and phase) are needed. A Graphical User Interface (GUI) will be made to let the user upload the files with the data and select some options, e.g. the number of generations and some specifications for the final response.

The workload is divided between the subgroups in the following way:

- **Mutator:** this subgroup will make children from the parents by mutating or reproducing the parents. Also making transfer functions that are used to evaluate each child will be done by the Mutator-group, just as making the embryo (initial) circuits.
- **Evaluation:** the evaluation subgroup will select some amount of children to become parents for the next round. This is done by a selection algorithm.
- **Controller:** the controller will make sure all communication between the other two groups is done properly. It will take care of designing and evaluating the cost function, removing components of the final circuit, but also making the GUI.

In Figure 1.1, a block diagram can be found where the signals with data between each subgroup have been drawn.

This thesis will be about the Controller subgroup. For the whole system to work, data has to be sent from one part of the program to another and every part needs to be executed in the right order. Also, in order for the Evaluation group to select certain children to become parents, a ranking must be made that indicates how good each set of filters is. To make this ranking, a cost function has to be designed that gives a score to how well each child behaves in the frequency range of interest. Lastly, users must be able to interact easily with the program using a GUI, such that the program can be used without changing the source code of the program or even needing to know how to program in Python.

2

Program of Requirements

The program of requirements (PoR) consists of two parts. The first part states the global requirements for the whole project, and the second part states the requirements that are specific for this subgroup and thus for the controller and the Graphical User Interface or GUI.

2.1. Global requirements

In this section, the requirements for the whole project are mentioned. This section consists of two parts: the mandatory requirements and the trade-off requirements.

2.1.1. Mandatory Requirements

Here, the mandatory requirements, i.e. the requirements that are at least needed to fulfill the goal of this project, are stated. First, the requirements for the overall system are given and then the requirements for the filters to be obtained. These requirements are formulated using SMART, which means that the requirements are Specific, Measurable, Assignable, Realistic and Time-related. The mandatory requirements are distributed over the three different subgroups and the subgroups add the time relation.

Program requirements

1. The program must take the frequency response of the individual drivers of a three-way loudspeaker system as an input.
2. The program must take the impedance of the individual drivers of a three-way loudspeaker system as an input.
3. The program must identify constraints for the filters based on the frequency responses and impedances.
4. The program must be able to design a passive analog filter for each driver with a minimal performance as specified in Section 2.1.1, Item 9.
5. The runtime of the program must not exceed 12 hours on an HP ZBook Studio G5 with an Intel Core i7-9750H CPU at 2.60 GHz.

Filter constraint requirements

6. The program must determine an operating range of each of the drivers, using the constraints from Section 2.1.1, Item 3.
7. The program must be able to design filters which only contain E12 series resistors (1Ω - $1M\Omega$), capacitors ($100pF$ - $100\mu F$) and inductors ($1\mu H$ - $1H$).
8. The analog circuits must filter out frequencies which are outside the operating range of the driver, found by the program.
9. The combination of filters must be able to create an acoustic frequency response of the loudspeaker system that is flat from 50 Hz to 20 kHz with a margin of 1.5 dB.
10. The analog circuits must not contain components which do not contribute to the requirements specified in Section 2.1.1, Item 9.

2.1.2. Trade-off requirements

The requirements in this section of the PoR would improve the usability of the final program. These requirements are not mandatory, but nice to have. The trade-off requirements, shortly ToR, are not SMART formulated, since it is not necessary to fulfill these requirements in order to have a working program. The ToR consist of three sets of requirements: for the program, the user and the filter.

Program requirements

1. The program should be able to design analog circuits for speakers systems varying from two to four speakers.
2. The program should be able to design active filters.
3. The program should indicate acceptable tolerances for components.
4. The program can find the monetary cost of components online.
5. The runtime of the program should be minimized.

User requirements

6. The user should be able to specify the amount of drivers in the speaker system.
7. The user should be able to set a different margin around the desired amplitude response than the standard 1.5 dB.
8. The user should be able to specify whether the program uses specific component values and/or a range of an existing E-series (e.g. E12) of component values.
9. The user should be able to specify a maximum number of components that can be used for designing the filters.
10. The user should be able to choose between passive or active filters for the final circuit.
11. The user should be able to choose the shape of the amplitude response of the complete system.
12. The user should be able to specify the monetary cost of components.
13. The user should be able to set a maximum total monetary for the final circuits, based on either component cost specified by the user or component cost specified by an online webstore.

Filter requirements

14. The group delay of the new acoustic response should be included in the cost function.
15. The attenuation of the each filter should be included in the cost function.
16. The combination of filters should have a response as close as possible to the shape of the pre-defined amplitude response.

2.2. Controller requirements

The Controller will be the brain of the algorithm, as explained in the introduction (Section 1.3). It will pass all data to the mutation and evaluation part of the program. Furthermore, a cost function will be created, where the user inputs will be taken into account and the operating range of the drivers will be determined. The requirements in this section are based on the global requirements that are stated before. All the global requirements are divided between the three subgroups.

2.2.1. Mandatory requirements

1. The controller must create a list for the Mutator containing the possible values of resistors, capacitors and inductors.
2. The Controller must make a cost function that scores a filter combination based on the deviation from a predefined acoustic frequency response between 20 Hz and 20 kHz. (Week 2)
3. The Controller must use the frequency response and impedance of a speaker to determine what the program will use as operating range. (Week 3)
4. The Controller must facilitate all communication between the evaluation and mutation classes. (Week 3)
5. The Controller must stop the program after either the circuit has converged to a minimum of the cost function or a predefined maximum amount of generations has passed. In an extreme case, the program will be stopped after 12 hours. (Week 3)
6. The Controller must remove components that are not needed to stay within the specified maximum amplitude deviation of the transfer function after the mutation/evaluation cycles have been finished. (Week 4)

7. The Controller must optimize the values of the final filter circuits. (Week 4)

2.2.2. Trade-off requirements

1. The cost function should account for constraints specified by the user.
2. The cost function should include group delay.
3. The cost function should include power usage of the filter circuits.
4. The Controller should find acceptable tolerances for the components, to stay within the required specification.
5. The Controller should find the monetary costs of components online.

2.3. GUI requirements

The GUI will allow users to easily use the program. This includes for example making it able for users to give their preferences as an input and showing the results graphically after the program is finished.

2.3.1. Mandatory requirements

1. The GUI must take the acoustic frequency responses of a three-way speaker as input. (Week 3)
2. The GUI must take the impedances of a three-way speaker as input. (Week 3)
3. The GUI must show the final optimized filter circuits after running the program. (Week 4)
4. The GUI must show the final total frequency response with the influence of the filter circuits. (Week 4)

2.3.2. Trade-off requirements

1. The GUI should allow the user to give inputs as specified in Section 2.1.2.
2. The GUI must be as intuitive and appealing as possible.

3

Controller

The controller makes sure that the other two subgroups have access to the required data in order for the functions to work properly. The controller also calls the right functions of the other subgroups in such a way that the algorithm is correctly executed. Apart from being a controller, a cost function will be designed and evaluated and some other functionalities will be implemented in the controller by this subgroup, e.g. determining the operating range of each driver and removing unnecessary components from the final circuits. The design choices of the controller and its functionalities will be explained in this chapter.

3.1. Speaker data

For the program to work, data of the drivers in the loudspeaker system is required. The program uses the magnitude and the phase of the acoustic response per driver in the speaker system and the magnitude and the phase of the electric impedance per driver for a frequency range from 20 Hz to 20 kHz. This range is a standard range, used in many applications for audio and it consists of exactly three decades. Having all this data makes sure that the transfer function of the filter can be calculated considering the non constant impedance of the driver. Additionally, the new acoustic response of the speaker system under influence of the filters can be calculated using the transfer function of the filters and the acoustic response of the driver. Both processes are described in Section 3.3. The data per driver is expected to be separated into two .csv files. The first file contains the data of the acoustic response, where the first column consist of the measured frequencies, the second column consists of the magnitude in dB of the acoustic response and third column consists of the phase of the acoustic response. The second file is expected to be the same structure, but for the impedance. Also, the magnitude should not be in dB, but in Ohms. The .csv filetype is easy to read in Python, so will therefore be used.

3.1.1. Logarithmically spaced data

Because of the way the data of the speaker was recorded, the difference in Hz between two sequential data points is constant. If this linearly spaced data was used, the higher frequencies have more influence on the cost function (Section 3.4) simply because there are more data points per decade for higher frequencies. To prevent this, a function was written that finds a predefined number of frequencies on a logarithmic scale between 20 Hz and 20 kHz. More data points result in a more accurate frequency response for both the filter and the acoustic response, but also a longer run time, since evaluating the cost function will take more time. 100 points gave a too low resolution, while having a lower runtime. For 1000 points, the resolution was higher than for 100 points and the runtime increased by a factor of approximately ten, while for 10.000 points, the resolution was again higher, but the also the runtime was again increased by a factor of ten. Taking this into account, the number of data points between 20 Hz and 20 kHz was set to 1000 points, since this is was the optimal trade-off between resolution and runtime.

3.1.2. Operating ranges

As explained in Section 1.1, drivers cannot produce an equally loud sound at every frequency. Depending on the physical parameters of a driver, it is more efficient in a certain range of frequencies. By combining multiple drivers into a single speaker cabinet, these ranges can together cover the complete band of all audible frequencies.

To create a flat acoustic response, it is important to use every driver mostly inside of its operating range. The speaker response is already more flat inside this range, which could guide the GA to use these ranges, but to help it, these ranges are used in the cost function, as is described in Section 3.4. Determining these ranges can be done with the driver data that the user provides. The lower boundary of an operating range is set one octave above the resonance peak in the impedance response of a driver [4]. An upper boundary could be established by comparing attenuation of the acoustic response of the loudspeaker measured at different angles, but this is not given as an input. Instead, for two drivers with adjacent operating ranges, the lower boundary of the higher driver is used as the upper boundary for the lower driver. For the speaker with the lowest operating range, the woofer, the lower limit of the operating range is fixed at 20 Hz, which is the lowest frequency the program will consider. Similarly, the driver with the highest operating range, the tweeter, has its upper limit at 20 kHz, the highest frequency considered.

3.2. Components

At the initialization of the `Controller` class, the E-series is given as a parameter to determine the possible values of resistors, capacitors and inductors. The use of E-series is preferred for a few reasons. First, using discrete values for components makes the problem space smaller, since there are less possible values, which is expected to result in a lower runtime, with the risk to find a less optimal solution. Also, if ideal values are used, rounding the final values to existing component values gives a result that is often worse than only using discrete values to find a solution [15]. Lastly, components can be bought in one of these series, which makes building the designed filters more easy. Considering this, E-series will be used to search for a solution. The default E-series will be the E12 series, which consist of 12 values per decade that are approximately equally spaced on a logarithmic scale. Most component series available to buy are E12 series. Changing the series per component is possible in the GUI (as requested by Section 2.3.2, Item 1).

Originally, resistors between 1 Ω and 1 M Ω , capacitors between 100 pF and 100 μ F and inductors between 1 μ H and 1 H were used, as specified in the program of requirements (Section 2.1.1, Item 7). In this way all the ranges consist of six decades of values, and all possible component values in the ranges can be bought. Currently, the range from 0.1 Ω to 10 k Ω is used for resistors, 1 μ H to 0.1 H is used for inductors and 1 nF to 100 μ F is used for capacitors. In practice, the resistors above 10 k Ω were too large, while resistances between 0.1 Ω and 1 Ω still can be used, since the impedance of the tested drivers lie between 3.5 Ω and 12 Ω . Inductors between 0.1 H and 1 H are difficult to manufacture. Often, when such inductors are made, the wires are thin or a metal core is needed to achieve the needed level of inductance. This results in non idealities as higher internal resistance and a non-linear inductance. Therefore, these values were omitted from the component value range. Capacitor values below 1 nF have a impedance higher than $1/(j2\pi * 20e3 * 1e-9) = -7958j \Omega$, which is in the same order of magnitude as the highest resistors. Values smaller than 1 nF will therefore have either too little or too much impact on the circuit and were thus also omitted from the original range.

3.3. Construction of the filter response

To evaluate the cost function, the transfer functions of all circuits of a child have to be determined. For the most part, this was done by the Mutator group [1]. After mutating the parents to create new children, the transfer functions of all circuits are determined. These transfer functions are calculated numerically, with two symbolic unknowns left in the function: the driver impedance and the complex frequency. These partly symbolic expressions are represented in Python using `SymPy`. To convert this expression into a function that handles arrays of frequencies and their corresponding driver impedance, the `lambdify` function is used [24]. `lambdify` returns a function that takes the arrays with the frequencies and impedances as an input and returns the filter response as an array of complex numbers.

The Evaluation group wanted to use this semi-symbolic transfer response for the selection procedure, which was the main reason why this approach was implemented.

Later on in the project, the Evaluation group did not need the filter response, and thus it was not necessary to use the previous method anymore. As will be explained in Section 3.5, the `lambdify` function is relatively slow and thus a second method to calculate the filter response is implemented. The Mutator group now receives two arrays with the complex frequencies and the impedance of the driver, and performs element wise vector operations to calculate the filter response. This results in an overall decrease in runtime of approximately 13 times, where both mutating the circuits and evaluating the cost function decreased in runtime.

To find the acoustic response with the influence of the filters, the following formula can be used:

$$AR_i = 20 \cdot \log_{10} \left| \sum_{j=1}^3 f_{i,j} \cdot a_{i,j} \right| \quad (3.1)$$

In this equation, AR_i is the i -th sample of the total acoustic response of the loudspeaker system, $f_{i,j}$ the i -th sample of the response of the filter for the j -th driver and $a_{i,j}$ the i -th sample of the acoustic response of the j -th driver. The filter response can thus be multiplied by the driver response, which gives the new combined response. These combined responses can be added together to create the total acoustic response. Any interference caused by a phase difference between two responses is accounted for, since all responses are still complex numbers. Because of this, it is also possible to subtract a response instead of adding it. This corresponds to connecting a speaker in reverse polarity, since a 180 degree phase shift in complex representation is done by multiplying by -1. With the new acoustic response of both the drivers and the loudspeaker system known, the cost can now be calculated using one of the cost functions.

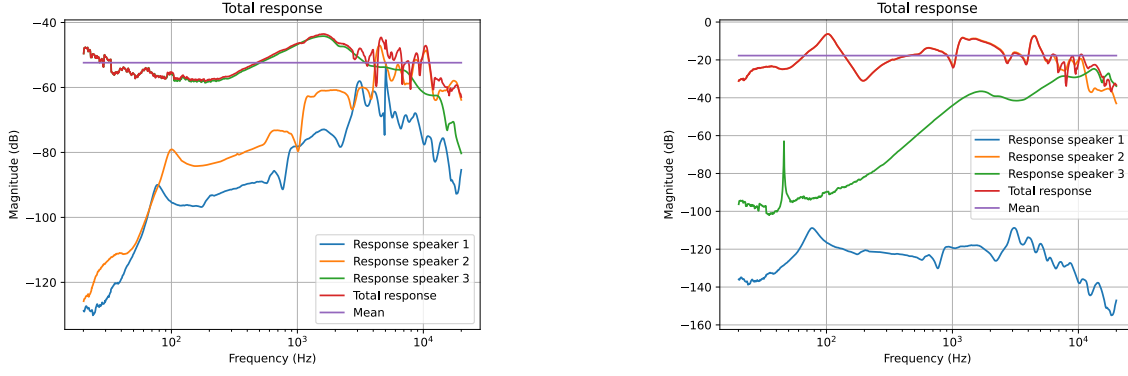
3.4. Cost function

In order to select certain children to become new parents, they need to be given a score so they can be ranked. This is done by a cost function that gives each child a cost. A child with a higher cost is worse than a child with a lower cost. To realize this, the cost function has to include all factors that make a combination of filters good or bad. As specified in the Program of Requirements in Section 2.2.1 Item 2, the deviation from a flat response must be included in this cost function. Furthermore, trade-off requirements (Section 2.2.2, Item 1-5) specify that power consumption, group delay, and requirements specified by the user should also be included.

There are multiple ways to implement a cost function. One option is to take all factors that should influence the cost of a child and quantify them. Then these different terms are added, each with their own weight. This weighted sum is then the total cost of a child. In this cost functions, the weights have two functions. First, they are used to correct any differences in the order of magnitudes of the terms. For example, one term might vary between 0 and 1, while another might vary from 0 to 1000, and to give them equal weight one of them must be scaled to match the order of magnitude of the other. The second function of weights is to give certain characteristics more emphasis than others. When a term adds more to the cost function, reducing this term has a bigger impact on the total cost than focusing on other terms. This can be used to help the program to reach the right objectives.

Another way to implement a cost function is to use the cost terms separately, instead of adding them together. This can be done by focusing on different aspects of children in multiple stages, where each stage uses a different cost. Using these costs, first certain characteristics are encouraged, and when these satisfy a threshold, the focus shifts to other characteristics. This approach has the advantage of allowing more direct control over the order in which traits are focused on, and when certain thresholds are satisfied. However, it does therefore require more tuning, since there are many different parameters, instead of simple weights.

For both methods, a cost function was developed. The design procedures of these cost functions are described in the next two sections.



(a) An example of the result with only the MSE of Equation (3.2) as cost function.

(b) An example of the result with the difference between the mean and a fixed power level of -20 dB added to the cost function.

Figure 3.1: The total response after running the program with the incomplete first cost function. The blue, orange and green lines are the woofer, mid-range driver and tweeter, respectively. The red line is the total response, and the purple line is its mean.

3.4.1. Weighted sum

First, a cost function was created using a weighted sum, consisting of five terms. Three of these terms are costs based on the separate driver responses, the other two are based on the total response. For the total response, the option of connecting a driver in reverse polarity to create a phase shift of 180 degrees was not considered for this first cost function.

The first cost term based on the total response is the mean squared error (MSE), of which the formula can be found in Equation (3.2).

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (Y_i - \hat{Y})^2, \quad Y_i = 20 \cdot \log_{10} |X_i|, \quad \hat{Y} = \frac{1}{N - M + 1} \sum_{j=M}^N |X_j| \quad (3.2)$$

In this equation, N is the amount of samples, which is 1000 in this case (as described in Section 3.1.1), Y_i is the magnitude of sample X_i in dB, \hat{Y} is the mean of the magnitude of the samples, M is the index of the sample where the frequency is closest to 50 Hz, and X_i and X_j are samples from the vector with the total complex response. The mean (\hat{Y}) was calculated from 50 Hz to 20 kHz, since the region from 20 to 50 Hz cannot be made equally loud as other frequencies with passive filters in most speaker systems, except by attenuating the whole response, which is unwanted. However, the MSE itself is calculated over the complete range from 20 Hz to 20 kHz, since even the response below 50 Hz should be made as flat as possible.

The second term is based on the attenuation of the total response. It is calculated by squaring the distance between the mean of the response above 50 Hz and a predefined acceptable power level, but no cost is added when the mean is above this level, as specified in Equation (3.3), where Att stands for attenuation. This acceptable power level is selected based on the lowest points of the acoustic response without filters, so that a flat response is possible for these low points, without having to amplify them. This cannot be done with passive filters.

$$\text{Att} = \begin{cases} (\hat{Y} - a)^2, & \text{if } \hat{Y} < a \\ 0, & \text{otherwise} \end{cases} \quad (3.3)$$

Here, \hat{Y} is again the mean magnitude of X_i in dB, as calculated in Equation (3.2), and a is the predefined acceptable power level in dB. This term was added to prevent power attenuation. Adding this to the cost function is only a trade-off requirement (Section 2.2.2, Item 3), but during early test runs it was discovered that without this term, the program would attenuate the acoustic response to inaudible levels. For example, in Figure 3.1a the mean of the total response is below -50 dB, while around -20 dB should be achievable, for the speaker system shown in Figure 1.2.

With only these two terms, the program has no incentive to use drivers inside their operating ranges, and it could be seen that it would use either the woofer or the mid-range driver for all lower frequencies.

For instance, in Figure 3.1b, the total response is almost exactly the same as the response of the mid-range driver (driver 2), while the tweeter and especially the woofer are strongly attenuated. To prevent this, the MSE and attenuation cost terms were also added for each driver response, inside their operating range. The MSE per driver is calculated as in Equation (3.4), and the attenuation per driver is calculated as in Equation (3.5).

$$\text{MSE}_{\text{drivers}} = \sum_{i=1}^3 \text{MSE}_i, \quad \text{MSE}_i = \frac{1}{N_i - M_i + 1} \sum_{j=M_i}^{N_i} (Y_{i,j} - \hat{Y})^2, \quad Y_{i,j} = 20 \cdot \log_{10} |X_{i,j}| \quad (3.4)$$

In this equation, \hat{Y} is the same mean as calculated in Equation (3.2). $Y_{i,j}$ is the magnitude of sample $X_{i,j}$ in dB, where i now stands for the driver number. Note that the number 3 in the first summation has to be changed for a speaker system with more or less than three drivers. M_i and N_i are the index of the boundary samples of the operating range of the i -th driver.

$$\text{Att}_{\text{drivers}} = \sum_{i=1}^3 \text{Att}_i, \quad \text{Att}_i = \begin{cases} (\hat{Y}_i - a)^2, & \text{if } \hat{Y}_i < a \\ 0, & \text{otherwise} \end{cases} \quad (3.5)$$

Here, \hat{Y}_i is now the mean magnitude of the i -th driver in its operating range and a is again the predefined power level.

Along with this addition, a fifth term was added to penalize using a driver outside of its operating range. This is not done by the other terms, since they only look inside the operating range of a driver. To achieve this, a region in which the driver should not be active had to be selected. This was done by taking every sample outside the operating range, with a margin of an octave above and below the operating range excluded as well. This term was implemented by Equation (3.6).

$$\text{MSE}_{\text{outside}} = \sum_{i=1}^3 \text{MSE}_i, \quad \text{MSE}_i = \frac{1}{N_i} \sum_{j=1}^{N_i} E_i^2, \quad E_i = \begin{cases} Y_{i,j} - (\hat{Y} - 10), & \text{if } Y_{i,j} > (\hat{Y} - 10) \\ 0, & \text{otherwise} \end{cases} \quad (3.6)$$

Again, \hat{Y} is calculated as in Equation (3.2). N_i is the amount of samples where the i -th driver should not be used, as explained above, and $Y_{i,j}$ is calculated as in Equation (3.4), for j as all samples where the driver should not be used. Using this equation, no cost is added if the response of a driver is attenuated by more than 10 dB outside its operating range. This threshold was chosen to allow a second order filter, with a slope of -12 dB per octave, without any cost, since the cost is calculated an octave from its operating range.

All terms then have to be added with their own weights. As mentioned earlier, these weights have two functions. First of all, it is important to scale all terms to the same order of magnitude. Since all terms in this cost function are based on squared differences of dB power levels, there are no large differences in orders of magnitude. However, the two terms based on the total response consist of one squared difference, while the other terms consist of three of these, since the costs for all separate drivers are added together. Therefore, the latter terms are three times as big as the former.

The other function of weights is to give some terms a bigger importance than others. Because the focus should be more on the terms based on the total response, their weights are increased. In addition, the weight of the $\text{MSE}_{\text{outside}}$ term is increased slightly to make sure drivers are not used outside their operating range.

The final weights for each term are visible in Equation (3.7)

$$\text{Cost} = 1.5 \cdot \text{MSE} + 1.5 \cdot \text{Att} + 0.2 \cdot \text{MSE}_{\text{drivers}} + 0.2 \cdot \text{Att}_{\text{drivers}} + 0.3 \cdot \text{MSE}_{\text{outside}} \quad (3.7)$$

3.4.2. Separate costs

The second cost function was designed by creating less cost terms, and using these costs in different stages of evaluation. For this cost function, three different costs are used, each in their own stage. In addition, elements of these costs are used to assign weights for selecting which filter should be

mutated, since only one of the three filters is selected for every mutation [1]. For example, if the cost of the tweeter is higher than the cost of the woofer, the probability of mutating the tweeter filter is increased, while the probability of mutating the woofer filter is decreased. This helps to make the mutations more effective, which decreases the amount of generations necessary. This is done by distributing every cost over three sectors, i.e. the three operating ranges.

The first cost used by this cost function is the cost per driver (CPD). This cost is a combination of the mean squared error and attenuation costs of the previous cost function, by defining the error as the difference between the response and the predefined target power level. Furthermore, the cost is first calculated as a 3x3 matrix (for a three-way speaker system): for every driver in every sector, a separate cost is calculated. These are added together per driver to create a cost per driver. Here, weights are used to give the cost in the sector corresponding to the operating range of a driver a bigger weight for that driver. This is formulated in Equation (3.8).

$$\text{CPD}_i = \sum_{j=1}^3 w_{i,j} \cdot \text{CPD}_{i,j}, \quad \text{CPD}_{i,j} = \begin{cases} \frac{1}{N_j - M_j + 1} \sum_{k=M_j}^{N_j} (Y_{i,k} - a)^2, & \text{if } i = j \\ \frac{1}{N_j - M_j + 1} \sum_{k=M_j}^{N_j} (E_{i,k} - a)^2, & \text{otherwise} \end{cases} \quad (3.8)$$

$$\text{with } Y_{i,k} = 20 \cdot \log_{10} |X_{i,k}|, \quad E_{i,k} = 20 \cdot \log_{10} |10^{\frac{a}{20}} + X_{i,k}|, \quad w_{i,j} = \begin{cases} 1, & \text{if } i = j \\ \frac{1}{2}, & \text{otherwise} \end{cases}$$

Here, i is the driver number and j is the sector number, a is the predefined target power level, and M_j and N_j are the index of the lower and upper boundary of sector j . $Y_{i,k}$ is the magnitude of sample $X_{i,k}$ in dB, and $E_{i,k}$ is the error sample $X_{i,k}$ adds to a flat line a in dB. This error is calculated by adding the complex response to a flat line, and comparing this to the flat line itself. This complicated expression is necessary, since the addition must be done before converting to dB, but the comparison should be done in dB. Finally, $w_{i,j}$ is used to make sure that the cost of the operating range of a speaker weighs as much as the total region outside the operating range. Since this region is split into two terms, they are multiplied by $\frac{1}{2}$.

In the second stage, the cost per sector is used (CPS), where the three sectors correspond to the three operating ranges. Here, the option of connecting one of the speakers in reverse polarity was accounted for. This gives four different options for a three-way speaker: no driver in reverse, or one of the three drivers in reverse. Two or three drivers in reverse does not change the relative polarity of the drivers. For all four connection methods, this cost was calculated, and the lowest was selected. This was done using Equation (3.9).

$$\text{CPS}_i = \frac{1}{N_i - M_i + 1} \sum_{k=M_i}^{N_i} (Y_j - a)^2, \quad Y_j = 20 \cdot \log_{10} |X_j| \quad (3.9)$$

In this equation, i is the sector number, and M_i and N_i the index of its boundary samples. Y_j is the magnitude of sample X_j of the total response, in dB, and a is the predefined target power level.

The third stage uses the maximal outlier of the total response, compared to the target level, calculated using Equation (3.10).

$$\text{Outlier} = \max |Y_i - a|, \quad Y_i = 20 \cdot \log_{10} |X_i| \quad (3.10)$$

Here, a is the predefined target power level, and Y_i is the magnitude of sample X_i .

In this cost function, frequencies below 50 Hz were ignored. Most woofers do not have a flat response from 20 to 50 Hz, and to create a completely flat response that extends down to 20 Hz, the complete response will be attenuated substantially. This is undesirable, and therefore, only frequencies above 50 Hz are examined.

When evaluating the children, the costs of each driver are added together to create a single total CPD. Similarly, a single total CPS is created. These are used to select children in their corresponding stage.

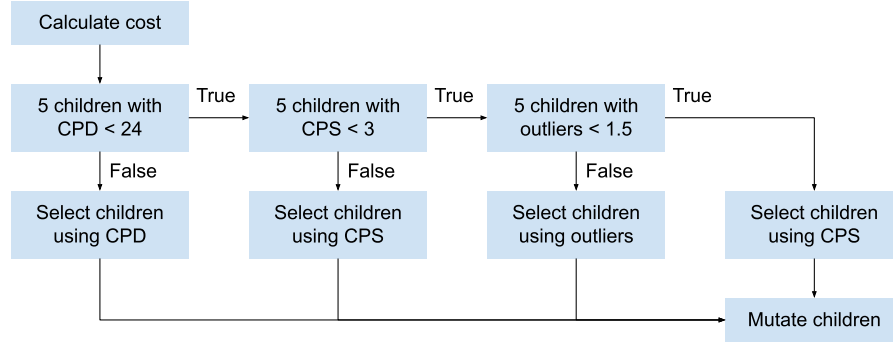


Figure 3.2: Block diagram of the selection procedure when using cost function 2.

When the program has to evaluate, the first stage is used by default. Only when at least five children satisfy the threshold of stage one, stage two is used, unless its threshold is also satisfied by at least five children. Stage three is only used if the thresholds of both the first and second stage are satisfied. This boundary of five children was selected, to make sure not only a single outlier satisfies a threshold, but at least a small part of the population. Since the stages were introduced to force the population to first focus on one characteristic, and then another, a single outlier should not allow the entire population to shift focus. It should be noted that for every selection moment, all checks are performed for which stage to use. This means that the program can frequently be seen switching between stages every couple of generations. This makes sure that when focusing on a certain cost, the other costs are not neglected. A block diagram of this selection procedure can be found in Figure 3.2.

The first stage uses the CPD, to make sure every driver is used mostly inside and only a little outside its operating range. Focusing on this cost does not guarantee a flat response, so it is only a first stage. The threshold for this stage is set to a CPD of 24. Since this is the sum of the separate costs, this corresponds to a cost of 8 per driver, although a lower cost for one driver can compensate a higher cost for another. This threshold was first set to 30, since it was observed that the decline of this cost started to slow down in that region. To prevent the program from spending much time in this stage, 30 was selected. Later on, the program became more efficient due to other improvements, which allowed to lower this threshold, without taking much more time. By not lowering the threshold further, more creativity is allowed in, for example, shifting the crossover ranges slightly.

The second stage uses the total CPS. In this stage, the flatness of the total response is mostly achieved, since the total CPS is an MSE of the total response, which is a good measure for flatness. For this stage, the threshold is set to 3. This was found to be a low enough cost that the total acoustic response is already quite flat, while it is still achieved in a timely manner, so that the program can move on to stage three.

Stage three uses the maximal outlier of the total response. The final requirement is that the response has to be within a margin of 1.5 dB (Section 2.1.1, Item 9), so when the majority of the total response is in this margin, the focus can be shifted to removing the remaining outliers. The threshold for this stage was set to 1.5, which corresponds to the aforementioned requirement.

When all three thresholds are satisfied, the program again uses the CPS to select children, since this cost was found to be the best at creating the most flat responses. Additionally, when the margin of 1.5 dB is satisfied, there is no need to focus on outliers anymore.

The use of these stages allows the evaluation to focus on different things throughout the runtime, depending on how good the current best children are. Moreover, it allows the evaluation to use these costs to determine which filter circuit is currently the worst per child, so the probability to mutate that filter can be increased. This is done by using weights for picking a filter to mutate.

For the first stage, the weights are set to the cost of each driver, since this is the exact cost used to select children. This means the program is more likely to select the driver with the highest cost, so that cost is most likely to decrease.

For the second and third stage, it is more difficult to use weights directly related to the costs. In stage two, the worst section is selected by looking at the costs per section. Then, the costs per driver in that section, as calculated in Equation (3.8), are used as weights. This is done to focus more on

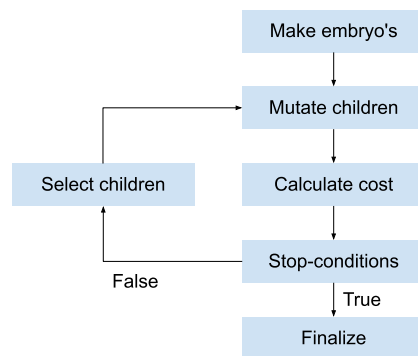


Figure 3.3: Block diagram of the main loop.

speakers that perform bad in this worst section.

Similarly, the third stage uses these costs per driver in a single section, but the chosen section in this case is the section where the maximal outlier is located.

3.5. Main loop

The main loop is the heart of the GA. It alternately calls the `Mutator` class to mutate the parents into new children, and the `Evaluation` class to select new parents from these children. When calling the `Mutator` or `Evaluation` class, all the data that these classes need are passed as arguments of the class `initializer` or their functions. The main loop repeats this process until one of the stop-conditions is met. A reason to stop the loop can be that the current generation is equal to the pre-defined maximal amount of generations, or that the maximum runtime has passed. A block diagram of the main loop can be found in Figure 3.3. After calculating the cost of all the children of the first generation, the child with the lowest cost is saved in a class attribute `best_child` in the controller. Since tournament selection is used to randomly select children to become parents, the best child of a generation may not be selected. Actually, the probability of not selecting the best child is around 20%, for a population size of 100, a selection size 30 parents and a tournament size of 5 [2]. For every generation after the first one, the cost of the child with the lowest cost of the generation is compared with the cost of the child that is saved in the class attribute. If the best child of the current generation is better than the child that is currently saved in the attribute, the new best child is saved in the attribute. This functionality was added to make sure that the all time best child will be shown to the user.

After running this loop for a few times with `lambdify` to calculate the filter responses, it became clear that calculating the filter response and evaluating the cost function costs a lot of time, over 2 times as much as creating the children. This is explained in more detail in Appendix B. The more complicated the circuits are, the more time it costs to evaluate them. So, the more generations are run, the more time is needed for evaluating the cost function. When splitting the cost function (the first cost function from Section 3.4) into different parts the time it takes to evaluate certain lines of code can be measured and printed into the python terminal. Calculating the mean or the MSE takes less than a millisecond, just as filling in the lambda function to calculate the filter response. Making the `lambdify` function however, takes typically tens of milliseconds and is the bottleneck in the cost function evaluation. Since `lambdify` was necessary in the early stages of this project, as described in Section 3.3, and it was the bottleneck in speed, it was not possible to make the function run faster. To still make the whole process of evaluating the cost function faster, making use of multitasking in Python was an option by using either multithreading or multiprocessing.

Multithreading from the `threading` module makes it possible for different threads to run concurrently [25], [26]. Two threads can run two different processes at the same time. Unfortunately Python has a Global Interpreter Lock or GIL [27]. The GIL makes sure that only one thread has control of the Python interpreter, which means that only one thread can execute a line of code at the same time. The processes on the thread will be executed at the same time, while the code in the processes will be executed sequentially. This means practically that adding threads will not increase the overall speed

at which the cost will be evaluated.

The second option, multiprocessing, makes it possible to execute multiple lines of code on different processing cores of the CPU and thus to run lines of code in parallel by getting around the GIL. The laptop from the PoR, Section 2.1.1, Item 5, has twelve logical cores, so in theory twelve different cost functions could be evaluated at the same time, resulting in a twelve times decrease of evaluation time. Using all twelve cores has the biggest decrease, but also has some downsides: the computer does not do anything else other than calculating the cost function. This results in some unwanted behaviour, including a frozen screen and not registering key presses. To prevent this behaviour from happening, the amount of cores not available for calculating was set to two cores, resulting in ten cores for calculating on the laptop. Using multithreading, the overall time it takes to evaluate all the cost functions for one generation had decreased evaluation time by a factor 3 (Appendix B). For multiprocessing the `ProcessPoolExecutor` class from the `concurrent.futures` library was used [28]. This specific library was easy to use. Only a few lines of code were needed to implement multiprocessing and the class works by starting a pool where processes (functions) can be added. The pool can then be executed in parallel. The rest of the Python script is not executed until all the processes in the pool are finished.

3.6. Removing components

As stated in [3], using trees to design circuits will result in many components that do not contribute substantially to the final transfer function of the filter response, or acoustic response for that matter. This might be caused by the fact that always three components with a certain value in a T-section are added to the tree. Apart from this case, it is also possible that two components are placed in series or parallel, where one of the components has a large value, and the other value is much lower. This results in either one of the components not contributing to the final acoustic response. To prevent spending more money than necessary, these components should be removed from the final circuit by replacing the component with either an open or a short circuit. The algorithm shown in Figure 3.4 was implemented to find which components can be removed and how. If removing a component resulted in decreasing the cost or increasing the cost slightly, the component will be removed. Different values for the maximum increase will be examined in Section 5.3. Having to buy less components decreases the monetary cost by some amount, while the decrease of the performance cost is little.

Calculating the cost of a short or open circuit is more difficult than was expected beforehand. To make an short circuit, the value of a resistor or a inductor can be changed to 0 and the value of a capacitor to infinity. For an open circuit, this is the other way around.

When the value of a component is changed to 0 or `SymPy.oo`, which is infinity in the `SymPy` library, the `lambdify` function does not know how to handle dividing by 0 or multiplying by infinity. It is possible to use the `simplify` function from `SymPy`, but this takes too much time for trees with more than a few T-sections. Changing the component type to a resistor and the value to 0 or infinity, removes the s or $j\omega$ in the equation for that component, but this method had no impact on the results. After the `lambdify` function was removed, the results were no different. Dividing by 0 and multiplying by infinity were no problem, but in the case that a component was removed, trying to remove a second component resulted sometimes in a new error. Removing some particular components resulted in $0 \cdot \infty$ ending up somewhere in the transfer function. Since this is not defined, the transfer function could not be calculated and `NaN` (Not a Number) was returned as the result of the function instead of an array with the filter response and cost cannot be calculated.

`lcapy`, the library used for drawing the circuits, as will be described in Section 4.4, has a function that calculates the transfer function from a netlist [29]. Unfortunately, this involves matrix inversion. For one T-section, this is doable, however having two T-section can result in a 10x10 matrix, which also takes too long to evaluate.

Another option is to run the simulation in SPICE and use that result to calculate the cost. This was not tried, since there was not enough time to implement this. Multiple functions need to be (re)written and it needs to be researched how it is possible to let Python initialize and run a SPICE program.

The last option was to change the value to near zero or near infinity and thus approximate the real values. Making the values either really small or really big (e.g. $1e-99$ or $1e49$) to approximate 0 or

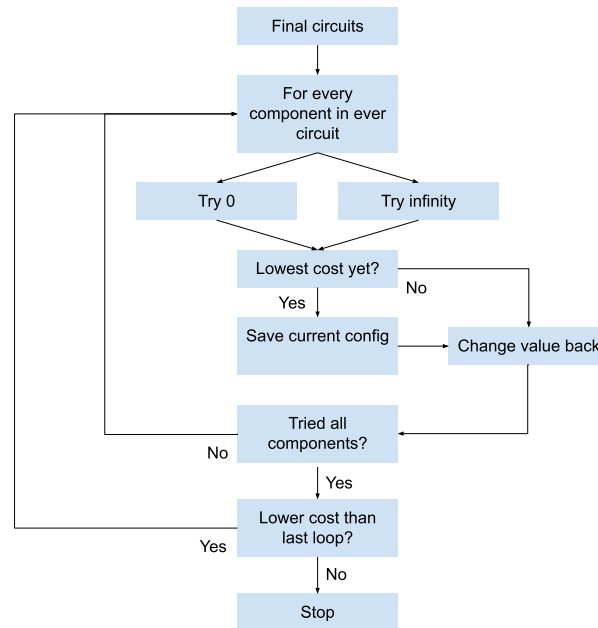


Figure 3.4: Block diagram of the algorithm for removing components. The value of every component is replaced by 0 and infinity and the new cost is calculated. After each component is tried, the value of the component where the cost was the lowest is changed. If no value change results in a decrease of cost, the loop is stopped.

infinity, gives more precise results. However, this may work for small trees, but for multiple T-sections with multiple removed components floats overflow because the exponent becomes too large. The lowest impedance using a component from the standard ranges and a frequency between 20 Hz and 20 kHz, is $1\ \mu\text{H}$, resulting in $20 * 2\pi j * 1\mu = j0.13\ \text{m}\Omega$. The highest impedance is the $10\ \text{k}\Omega$ resistor. $1\text{e-}10$ is used to approximate 0 and $1\text{e}10$ to approximate infinity. These value are respectively $1\text{e}6$ smaller or bigger than the lowest or highest impedance, but small enough that overflow will most likely not occur.

3.7. Last value optimization

Since some components are removed by the function described in the previous section, the optimal value of the components that are still in the circuit can be a little bit sub-optimal. Also it is likely that not all the values for the different components are tested. The Mutator chooses a mutation randomly, where only one of the mutations changes the value of a randomly chosen component. It is assumed that the value of every component is in the neighborhood of the optimal value and with this assumption, the algorithm described in the block diagram found in Figure 3.5 was constructed. This way a local minimum is found for every component.

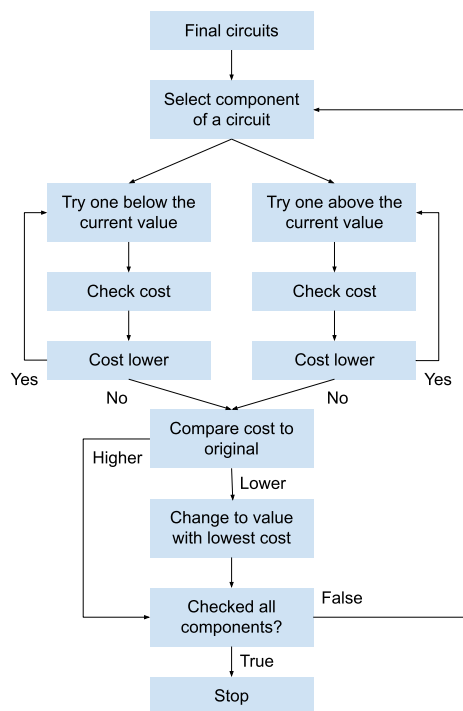
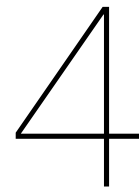


Figure 3.5: Block diagram of the algorithm for the last value optimization. For every component, the current value of the component is replaced by the value in the E-series below the current value and the cost function is evaluated. If the cost is lower, the next lower value is tried. If there is no more decrease in cost, the loop is stopped. After the same is done for the next value above the current value, the lowest cost of the lower and higher value are compared and the value of the component is changed to the value with the lowest cost.



GUI

In order for the user to be able to easily interact with the program in an intuitive way (Section 2.3.2, Item 2), a Graphical User Interface or GUI is made. In this GUI, the user can select data files to load and change the available settings with knobs and in text fields, without the need of changing the settings directly in the source code. In this way, no Python (or programming) skills are needed in order to use the program. In this chapter of the thesis, the design choices for the GUI are stated and clarified.

4.1. Python library for GUIs

As stated before, the program is written in Python and so is the GUI. There are quite some libraries for Python available that can be used for designing and building a GUI, where `TKinter` and `PyQt` are most commonly used. `TKinter` is a built-in Python library containing many widgets. It is often combined with `TTk`, or Themed Tkinter library, since `TTk` widgets have a more appealing look. Both libraries have extensive documentation and are easy to use, thanks to the many online examples of the widgets [30]. `PyQt` is more professional and is more a framework than a library. There is a GUI builder, where widgets can be dropped and the python code is generated. Also it has more functions, e.g. the framework can plot graphs by itself, while for `TKinter`, another library is needed. On the other hand, the framework is less well documented, harder to use and a licence is required to use it for a commercial application [30]. Considering all this, it was chosen to work with `TKinter`, since it is a little easier to start with. It may look a little less professional, but this is not a problem. The focus will mainly not be on the GUI, but more on the controller and its functionalities. Also, the fact that `TKinter` has better documentation and one of the authors has already used this library before, had an influence on the decision.

With the library chosen, the rest of this chapter will go into further detail about the design choices of the GUI.

4.2. Loading files

As stated in Section 2.3.1 Items 1 and 2, the GUI must take the acoustic response and the impedance of a three-way speaker. As described in Section 3.1, the data is expected to come into a total of six files (three files containing acoustic data and three files containing the impedance; two files for each speaker). So, there are six fields placed on the main GUI window, where the files can be selected, as can be seen in Figure C.1. Also a `ttk.Checkbox` was added to select the amount of drivers in the system. Selecting a different value than the default of three, results in fields appearing or disappearing where files can be selected. After the files are selected, the load button can be clicked to initialize the `Controller` class and load the data into the class. After the data is loaded, the acoustic response per driver and the total acoustic response are plotted below the file fields. An example of how the GUI looks like at that moment can be seen in Figure C.3

Now the maximum amount of generations can be entered in the empty `ttk.Entry` and the run button can be clicked to start the main loop (Section 3.5) and to start the process of designing the filters.

4.3. Multithreading

When running the main loop or loading the data, the GUI would freeze, due to the GIL. Until a line that explicitly requests the GUI to update, e.g. `GUI.update()`, the GUI would stay frozen. It would then unfreeze to update the GUI, to get frozen again after the line was fully executed. Resizing the GUI or even closing the GUI would not happen, until the update was requested and executed. To improve the user experience and make the GUI more responsive and appealing (as required in Section 2.3.2, Item 2), multithreading was used, as explained in Section 3.5. In this case, executing processes sequentially instead of parallel is not a problem, since the GUI has to be updated only when the user asks for it. Running the GUI and the main loop in two threads will make the main loop a few percent slower, since some lines of GUI code will be executed during the main loop process. Working with a frozen GUI is difficult and makes it less appealing and the decrease in speed is minimal, so this trade-off was accepted and multithreading the GUI and the main loop was implemented.

4.4. Showing results

The PoR describes that the final circuits, one for each driver, should be shown to the user (Section 2.3.1, Item 3), just as the final calculated acoustic response under the influence of the designed filters (Section 2.3.1, Item 4). As explained in Section 3.5, the overall child with the lowest cost is saved in a class attribute in the `Controller`. This child consists of a list of three `Tree` classes (for a three-way speaker). The tree-structures are used to build the circuits using the `lcapy` library. `lcapy` makes use of the `Circuitikz` package from \LaTeX to draw circuits. `lcapy` can be used to analyze circuits and make a netlist of a circuit to draw the circuit. Unfortunately, letting `lcapy` draw the circuit resulted in many overlapping wires or drawing two components on top of each other. Especially the parallel connected T-sections resulted in these kinds of problems.

To solve this problem, a class is written, `Circuit`, that recursively builds the netlist, T-section by T-section, and makes sure that there are no overlapping wires or components. Another advantage of not using a function from an existing Python library to design a netlist, is the extra flexibility. It is easier to replace components by a short or an open circuit (Section 3.6) or change the value of components (Section 3.7).

An example of a final acoustic response and a circuit can be seen in Figure 4.1 and Figure 4.2 respectively.

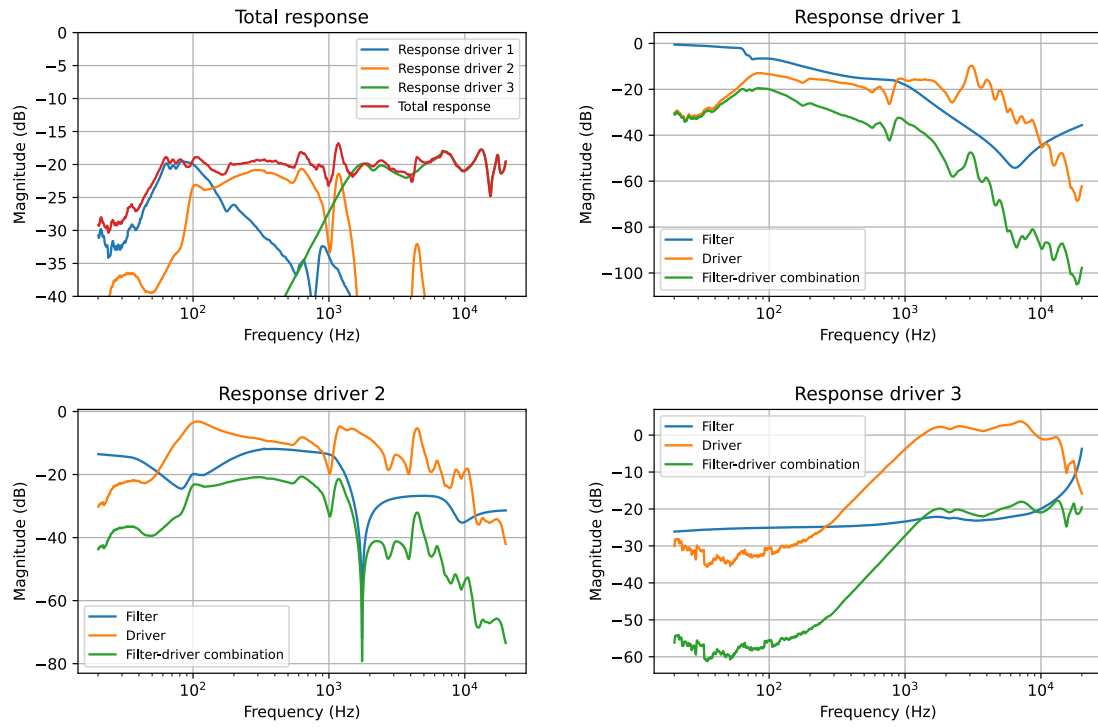


Figure 4.1: An example of a acoustic response under the influence of the designed filters. The top left figure shows the acoustic response of each speaker influenced by the filter and the total acoustic response. The other three figures show the original acoustic response, the filter response and the combination of the two per driver.

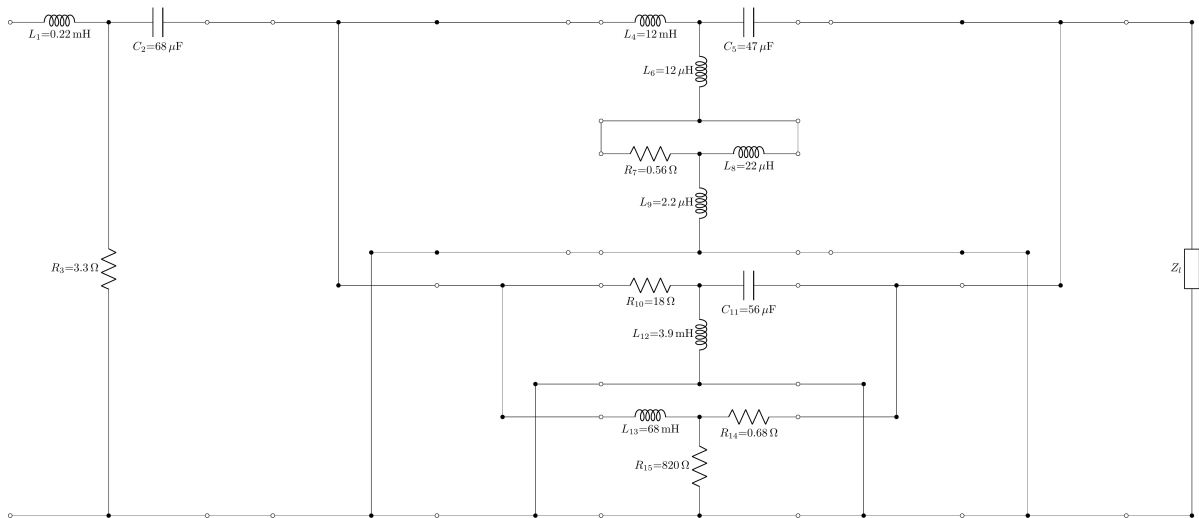


Figure 4.2: An example a designed circuit. At the left two ports the voltage source (audio) needs to connected and at the right the load (driver) is connected, currently symbolized by the impedance Z_L .

5

Implementation, Validation and Discussion

In this chapter, the obtained results will be validated and explained. This chapter consists of three parts: first, the process in which all the solutions of the design process are implemented is explained. Then, the results of the Controller subgroup will be validated, as well as the final product itself. Finally, the results are discussed.

5.1. Implementation process

With the design choices described in the previous two chapters about the controller and the GUI, different functions were implemented. This was mostly done using a trial and error method. A function was first implemented in the way according to a design choice and then improved if necessary or rewritten if the idea did not work. Making the design choices and implementing the choices was an iterative and intertwined process, where changing or updating the choices according to experience with the current implementation was done if it seemed necessary.

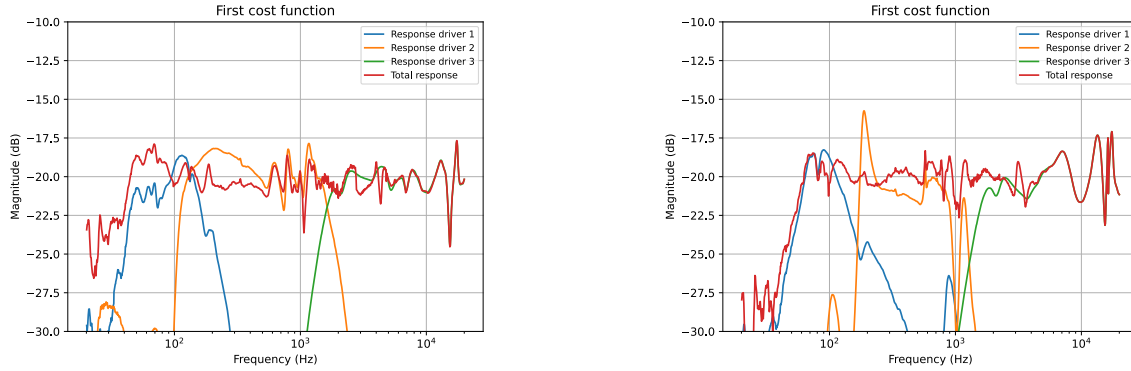
5.2. Cost function validation

To compare the two implemented cost functions, both were used in multiple runs for 1000 generations, which takes about 20 to 30 minutes. Since both cost functions calculate different costs, it is difficult to compare these. Instead, for the best children found with the first cost function, a cost was also calculated with the second cost function. For this comparison, the total CPS was used, as calculated in Equation (3.9). This gave the most fair comparison. Since the first cost function contains many terms, and some of these terms are not included in the second cost function, the results found when using the second cost function have a much higher cost according to the first cost function. This is not the case when using the CPS, since this cost is implicitly included in terms of cost function 1. The results of this comparison can be seen in Table 5.1, and the best results of both cost functions in this run are shown in Figure 5.1.

After running the program three times for both cost functions, the performance differences are small. As mentioned in Table 5.1, the average total CPS for the results obtained with the first cost function was

Table 5.1: Results of comparing the different cost functions. For each cost function, three separate runs of 1000 generations were done. Then, for the best child of that run, the CPS was calculated using Equation (3.9)

Run	CPS of results when using cost function 1	CPS of results when using cost function 2
1	2.3744	4.7646
2	4.8121	2.8460
3	3.1851	2.9224
Average	3.4572	3.5110



(a) An example of the result using cost function 1. This result corresponds to run 1 of cost function 1 in Table 5.1.

(b) An example of the result using cost function 2. This result corresponds to run 2 of cost function 2 in Table 5.1.

Figure 5.1: Examples of the total response after running the program with both cost functions. The blue, orange and green lines are the woofer, mid-range driver and tweeter, respectively. The red line is the total response.

3.4572, while for the results obtained with the second cost function, this was 3.5110. Comparing this difference to the differences between the results, it is clear that this difference is insignificant, and more tests are necessary to make a conclusive decision about which cost function is better. Unfortunately, doing these test was not possible due to a lack of time. However, it does show that randomness is a big factor in the program, at least when running it for 1000 generations. It should also be noted that both cost functions could be made better by using more time and data to tune their weights or parameters.

Even though the first cost function produced a slightly better average and the best overall result (Figure 5.1a), cost function 2 was selected for this program. Multiple reasons led to this decision. First of all, the second cost function gives more insight in how good the filter combinations are at different aspects. Moreover, with this cost function, it is already determined whether a child satisfies the 1.5 dB margin, i.e. when the maximal outlier is below 1.5. Second, because of the stages cost function 2 uses, there is more certainty on what a low cost means. For the first cost function, a low cost could mean a good performance per driver, but a less optimal overall score, or vice versa. This uncertainty complicates the process of selecting the best child. Finally, the second cost function is expected to outperform the first cost function in longer runs. This is due to the stage system, which gives a more directed way of reducing the cost. It is worth noting that after 1000 generations, only one of the three runs with cost function 2 had reached stage 3.

5.3. Removing components validation

As described in Section 3.6, the cost is allowed to increase a little bit when components are removed, to stimulate the removal. Maximal increases of 1%, 5% and 10% were tried, of which the results can be found in Tables 5.2 and 5.3. On average, 54 components are removed and the value of 21 components is changed. A total of 75 components is either removed or changed in value on average. With a maximal increase of 5%, the most components are removed, but this is mostly due to the outlier in run 7, where 80 components were removed. Overall, the decrease in cost was the highest for 10% with 11.3%. Although the decrease for 1% is lower (9.85%), the decrease is more consistent than for 10%. Also, the mean of the original cost after 1000 generations is lower for 1% than for 10%. Since it is more difficult to decrease a lower cost, the difference between the two decreases is less significant. Taking all this into account, the threshold of 1% was selected.

Currently, a tree can consist of at most 16 T-sections. Since there are three components per T-section and three trees per child, a total of $16 \cdot 3 \cdot 3 = 144$ components can be used per child. This means that about a third of all the components is removed and almost half of the components is either removed or has its value changed for a maximal increase of 1%.

When components are removed, in the circuit the component is replaced by either just a wire or no wire. This results in many dangling wires that are not connected anywhere, as can be seen in Figure 5.2. While the schematic is correct, it is hard to read correctly.

Table 5.2: Number of components that are removed (made open or short) and where the values was changed. The data was gathered by running the program with a population size of 200, 70 parents and a tournament size of 4. The program run for 1000 generations and the cost was calculated using the second cost function.

Run	Max cost increase	Number of components removed	Number of values optimized	Cost after 1000 generations	Cost after removing	Cost after optimizing
1	1%	48	36	2.7701	2.7950	2.4060
2	1%	54	27	5.2791	5.3222	4.9138
3	1%	51	18	2.8252	2.8216	2.4162
4	1%	30	37	4.8967	4.9451	4.2933
5	1%	35	20	4.8544	4.8888	4.7390
6	5%	49	15	3.2636	3.4244	3.2881
7	5%	80	14	3.2613	3.4204	2.8772
8	5%	58	15	4.0495	4.2330	3.8562
9	5%	53	13	6.3290	6.6328	5.5886
10	5%	58	16	5.2685	5.4573	4.9190
11	10%	61	25	3.7284	3.9841	3.0945
12	10%	65	24	3.205	3.502	2.9303
13	10%	60	18	5.5438	5.9842	4.453
14	10%	52	21	6.3743	6.9003	6.0695
15	10%	52	19	5.0161	5.4929	4.6798

Table 5.3: Averaged data from Table 5.2 per maximal cost increase percentage. The average increase after removing components and the average decrease after optimizing are both relative to the average original cost.

Maximal increased cost	Average number of components removed	Average number of values optimized	Average cost after 1000 generations	Average increase after removing	Average decrease after optimizing
1%	43.6	27.6	4.1251	0.657%	9.85%
5%	59.6	14.6	4.4344	4.54%	6.83%
10%	58	21.4	4.7735	8.37%	11.3%

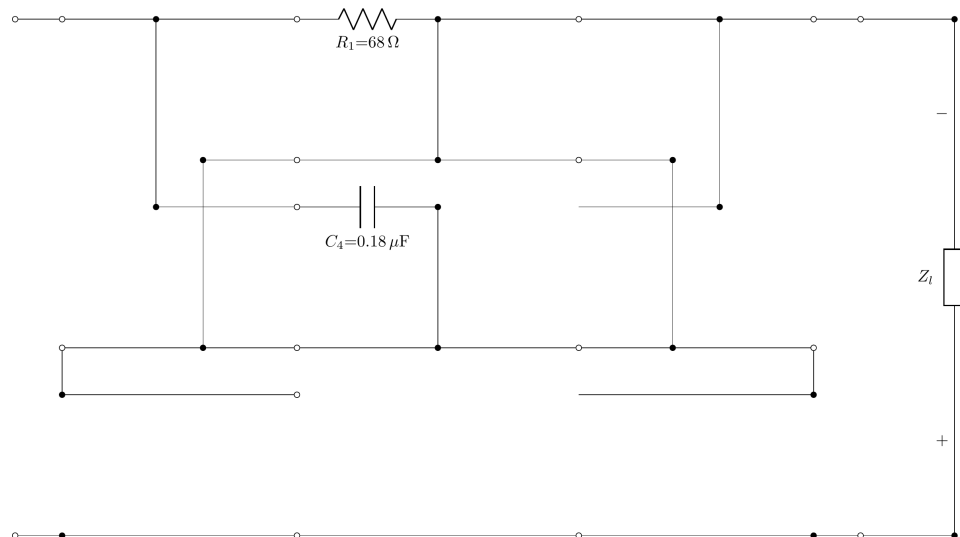


Figure 5.2: An example of a circuit after removing all the components. There were 7 of the 9 components removed, resulting in a lot of dangling wires. Actually, this circuit is equal R_1 and C_4 being in parallel and the RC combination being in series with the driver.

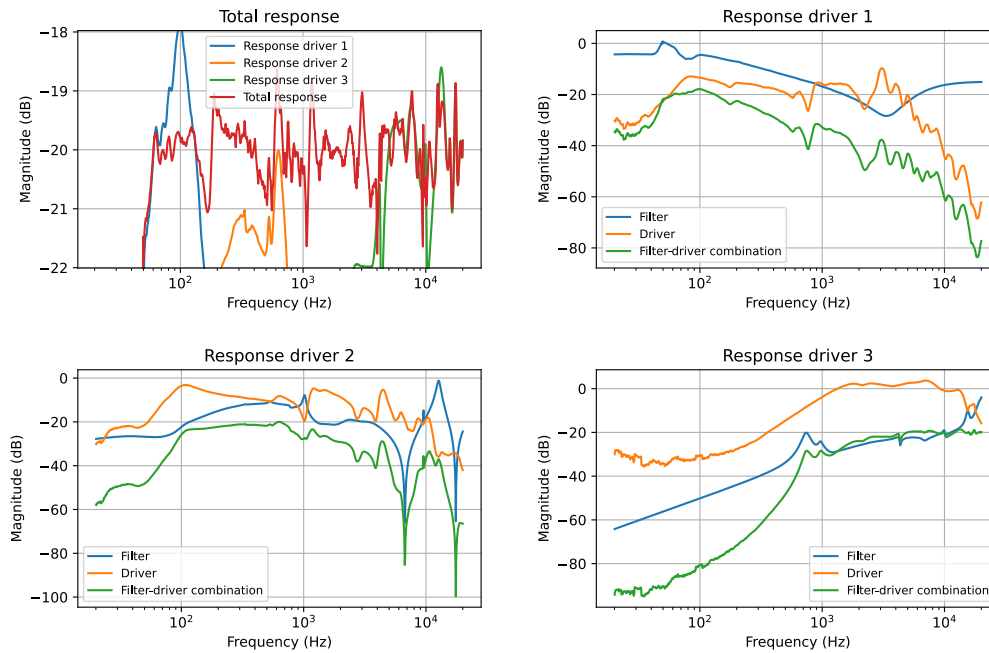


Figure 5.3: The best result found, after removing components and the last optimization.

5.4. Last value optimization validation

The last optimization was done, to optimize the components one more time after components are removed (Section 3.7). As can be seen in Tables 5.2 and 5.3, doing the last optimization works and reduces the cost with 9.34% relative to the original cost after 1000 generations. Relative to the cost after the components are removed, the decrease is even higher: 13.2%. The data shows that even when the cost increases to remove more components, most of the time (in 14 of the 15 runs), the cost is lower after the last optimization than it was after running the 1000 generations. In the end, a decrease of around 10% can be expected relative to the cost before removing the components, since the maximal increase of 1% will be used.

5.5. Product validation

With the Mutator and Evaluation group also having their parameters tuned and optimized, a few runs were done to measure the performance of the total program. The best result of these runs can be found in Figure 5.3. The rest of the figures, i.e. the plot of the lowest cost per generation and the zoomed out figures of the acoustic response, and the circuits, can be found in Appendix D. As can be seen in the figure, except for a few outliers, most of the response falls within the 1.5 dB boundary from Section 2.1.1, Item 9. There are few outliers bigger than 1.5 dB, with a maximal outlier of around 1.8 dB.

5.6. Discussion

All of the mandatory requirements from the PoR of the Controller and GUI (Sections 2.2.1 and 2.3.1 respectively) are satisfied. Most of the trade-off requirements however, are not fulfilled. In fact only one of them is fully fulfilled: Section 2.2.2, Item 3. Section 2.2.2, Item 1, and Section 2.3.2, Items 1 and 2 are partially fulfilled. The constraints that are used in the controller or cost function can be set by the user, but not all the user constraints from Section 2.1.2 are used. With a additional week, it is expected that most of the trade-off could be implemented. The GUI is, in our opinion at least, reasonably intuitive, but not appealing. This has mostly to do with the fact that the focus during this project was more on the controller and its functionalities than the GUI.

6

Conclusions

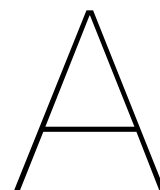
This thesis has discussed the design and implementation of a controller and GUI for an AI loudspeaker filter design program. For the controller, two different cost functions have been designed. One used a single cost, a weighted sum, which was minimized. The other used different costs that were separately minimized in different stages. Comparing these, no significant difference was found, although there was not enough data for a conclusive decision. In order to remove unnecessary components after the final circuits were found, multiple strategies were tried to create a working algorithm. In addition, an algorithm was designed to optimize all component values as a final step after removing components. These algorithms were then used to compare different maximal allowed increases in cost when removing components. This showed that there is a trade-off between the number of components and the final cost of the circuits. In the GUI, the acoustic response and impedance of multiple drivers can be given. After the program has executed, the final acoustic response using the best filters is shown, together with the circuits of these filters.

To create a working program, the controller calls mutation and selection functions in a loop, together with its own functionalities. This program gives flat responses within 1.5 dB, except for a few outliers.

6.1. Future work

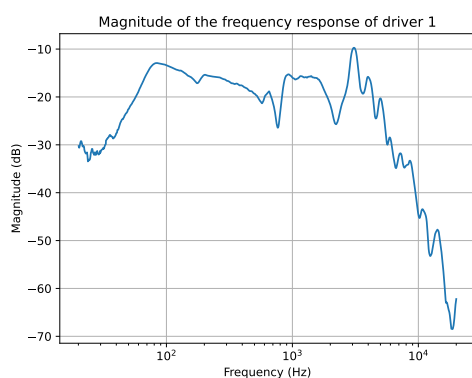
A few things can be done to improve the final product. For example, the circuit that is shown in the GUI after the design is finished has dangling wires, when components are removed. Implementing a way to remove those wires and simplify the circuit representation would improve the user experience and the ease of physically realizing these circuits. It could also be tried to run some calculations on the GPU instead of the CPU. The GPU has many more cores, making it possible to do more calculation in parallel, possibly decreasing the runtime. Improving the GUI would be another point where improvement is possible. The GUI works, but is not very appealing. Putting some more attention in the appearance of the GUI would be beneficial. Lastly, implementing more trade-off requirements would also improve the quality of the product.

In future research, more data should be acquired to compare both cost functions, and more investigation could be put into cost functions and tuning the weights and other parameters. Also, the responses of different loudspeaker systems should be measured to see whether the program also works for other systems. It is likely that some parameters need to be tuned again. Additionally, more algorithms for removing components and the last optimization could be further researched. Right now, components are removed by making the value relatively small or big. Simulating the results in SPICE, where the components are actually removed, might improve the accuracy. Furthermore, by using more complex algorithms from literature, the performance of both algorithms is likely to be improved.

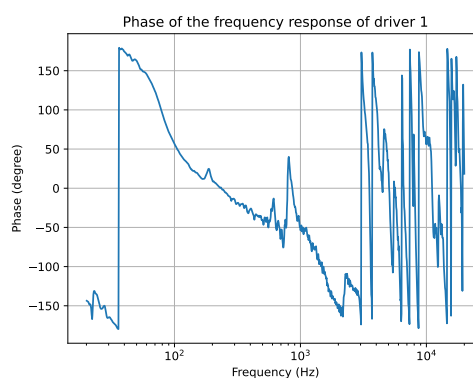


Figures of test speaker

In this appendix, plots with the data of the drivers can be found. The EPO-1 loudspeaker consists of two bass drivers, one mid-range driver and one tweeter. The measured frequency response in Figure A.1a is for one bass driver, while the impedance measurements (Figure A.1b) is done with the two bass drivers connected in parallel. To account for the the second bass driver in the frequency response, 6 dB can be added and the impedance can be divided by two to get the impedance of one woofer. In Figures A.3 and A.4 and Figures A.5 and A.6 the frequency and impedance plots of the mid-range driver and tweeter can respectively found.

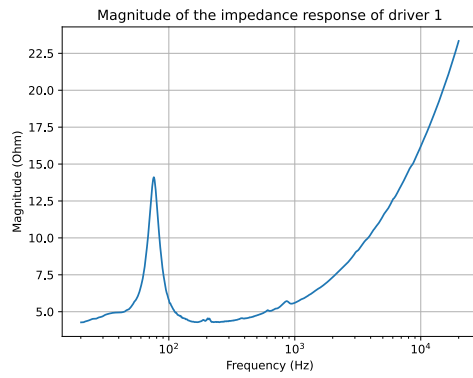


(a) The magnitude plot

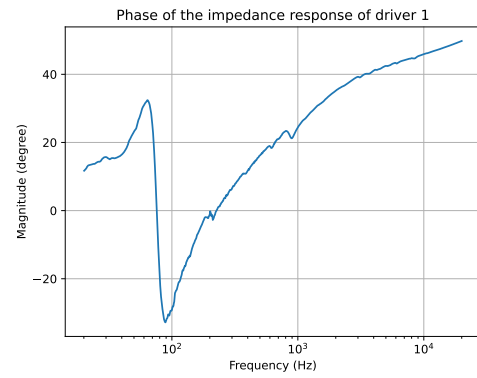


(b) The phase plot

Figure A.1: The magnitude and phase plot of the acoustic response of a single bass driver of the EPO-1 loudspeaker system

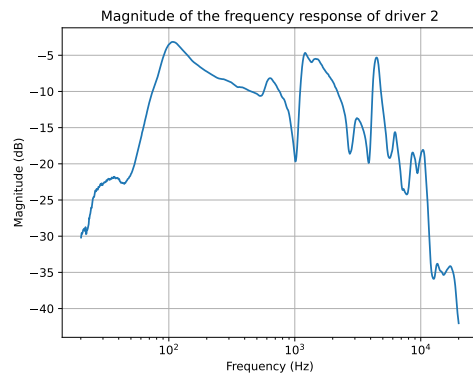


(a) The magnitude plot

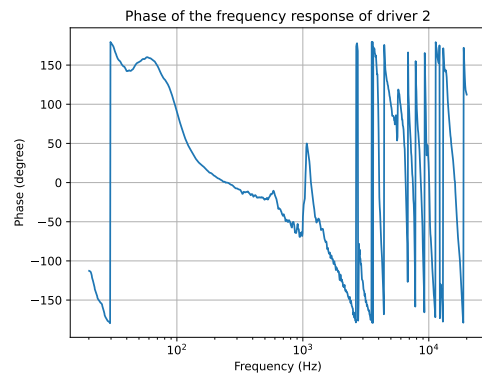


(b) The phase plot

Figure A.2: The magnitude and phase plot of impedance of two bass drivers of the EPO-1 loudspeaker system in parallel

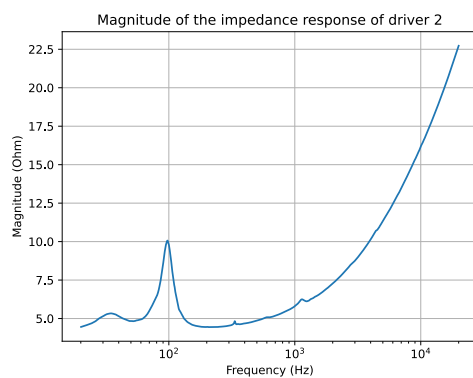


(a) The magnitude plot

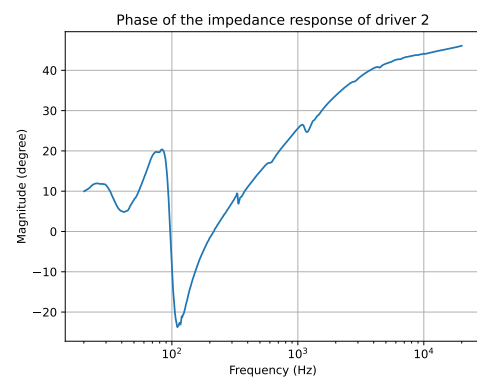


(b) The phase plot

Figure A.3: The magnitude and phase plot acoustic response of the mid-range driver of the EPO-1 loudspeaker system

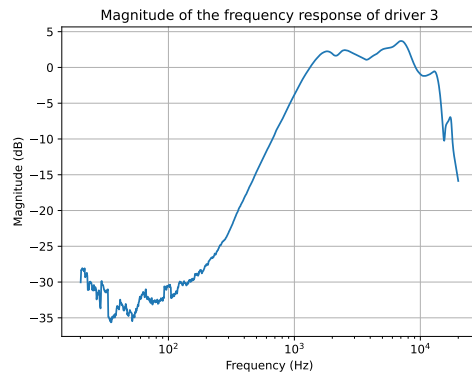


(a) The magnitude plot

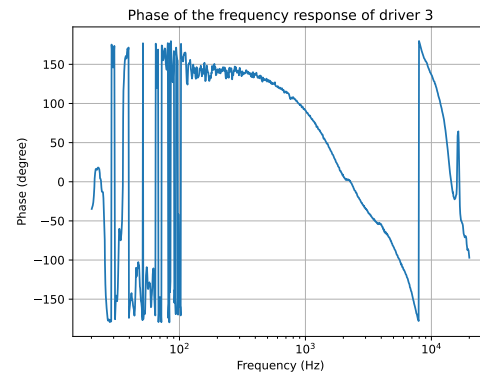


(b) The phase plot

Figure A.4: The magnitude and phase plot of impedance of the mid-range driver of the EPO-1 loudspeaker system

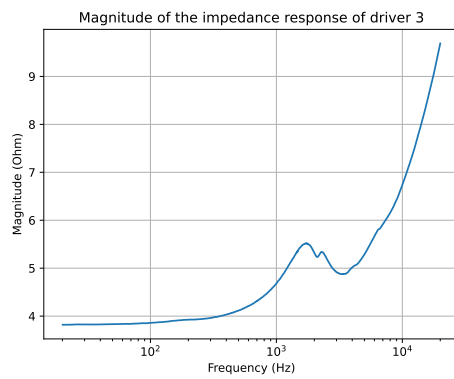


(a) The magnitude plot

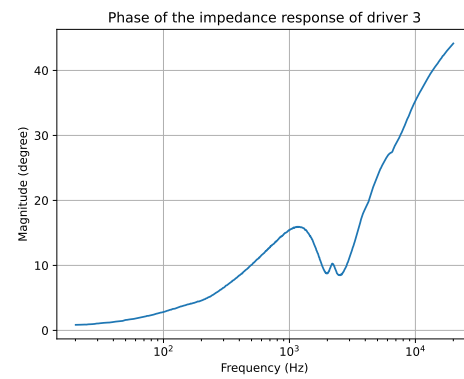


(b) The phase plot

Figure A.5: The magnitude and phase plot of the tweeter of the EPO-1 loudspeaker system



(a) The magnitude plot



(b) The phase plot

Figure A.6: The magnitude and phase plot of impedance of the tweeter of the EPO-1 loudspeaker system

B

Computing time of the cost function

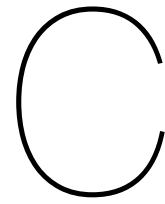
As explained in Section 3.5, evaluating the cost function is computationally expensive and costs relatively much time, especially using the `lambdifly` function to calculate the filter responses. To decrease the total time it takes to run the program, multiprocessing is used, as explained in Section 3.5. The time it takes to evaluate different parts of the main loop using one core and ten core multiprocessing, can be seen in Table B.1 and Table B.2 respectively. The main difference between the numbers in the two tables is the time it takes to evaluate the cost function: 137 seconds on average with one core and 76.4 seconds on average with ten cores. This means that multiprocessing on the laptop from the PoR results in a 1.79 times decrease of runtime for evaluating the cost function and a 1.43 times over-all decrease of runtime. Worth mentioning is the last row of both tables, indicating the time it took to complete the second half of the 50 generations, i.e. generation 26 to 50. The decrease in cost function evaluation and over-all runtime are respectively 2.95 times and 2.02 times. This indicates that the larger or more complicated the circuits are, the larger the decrease in runtime is. This can be explained by the fact that a larger circuit cost more time to evaluate and the time to start the multi-core process can be averaged out over a longer period of time.

Table B.1: Calculation time used to run the main loop for 50 generations, with a population size of 100, 30 parents and a set size of 5 and the `lambdifly` function. Run on HP ZBook Studio G5. Evaluating the cost function uses one process.

Run	Total time (s)	Total mutation time (s)	Total cost function time (s)	Total selection time (s)
1	76.0	40.2	31.3	4.52
2	289	69.0	206	13.9
3	250	62.6	176	12.2
4	166	54.0	103	9.54
5	266	80.3	170	15.8
Avg.	210	61.2	137	11.2
Avg. gen 26 - 50	152	39.0	104	8.5

Table B.2: Calculation time used to run the main loop for 50 generations, with a population size of 100, 30 parents and a set size of 5 and the `lambdifly` function. Run on HP ZBook Studio G5. Evaluating the cost function ten parallel processes.

Run	Total time (s)	Total mutation time (s)	Total cost function time (s)	Total selection time (s)
1	152	68.5	74.2	9.35
2	99.7	42.5	51.8	5.38
3	136	60.4	68.7	6.32
4	240	96.4	132	11.1
5	110	48.9	55.0	6.00
Avg.	147	63.3	76.4	7.64
Avg. gen 26 - 50	75.1	35.7	35.3	4.14



GUI Windows

How the GUI looks in different stages of the program can be found in this appendix. This appendix will consist of figures of the windows with a short explanation of at what stage the program is and what the GUI shows.

C.1. Start window


When the program is started, the window from Figure C.1 will appear. The amount of speakers and the E-series per component type can be select and the .csv file locations can be loaded.

C.2. Loading data

When the files and the right E-series are selected, the load button can be clicked to initialize the `Controller` and load the data into the class. During loading the data, the GUI looks like Figure C.2. After loading the data is done, the frequency responses of all the speakers are plotted, together with the total acoustic response, without any filters. The GUI looks then like Figure C.3.

C.3. Running the program

When the maximum amount of generations is entered and the run button is clicked. A progress bar appears that keeps track of the amount of generations that are finished (Figure C.4). When the last generation is executed, the results are plotted: for each filter the circuit that should be build is shown (e.g. Figures C.5 to C.7), just as a figure with the new acoustic response (both for the separate driver and the total of the cabinet) and per driver the measured acoustic response, the filter response and the acoustic response under influence of the filter is plotted in Figure C.8. Lastly, also a graph with the minimal cost per generation is plotted (Figure C.9).

 SOWAL

Select the amount of speakers:

3

⌵

Load data

Select frequency/ response of speaker 1:

Choose a file

Browse

Select frequency/ response of speaker 2:

Choose a file

Browse

Select frequency/ response of speaker 3:

Choose a file

Browse

Amount of generations:

⌵

Run

Select impedance of speaker 1:

Choose a file

Browse

Select impedance of speaker 2:

Choose a file

Browse

Select impedance of speaker 3:

Choose a file

Browse

Settings

Resistor E-series: E12

⌵

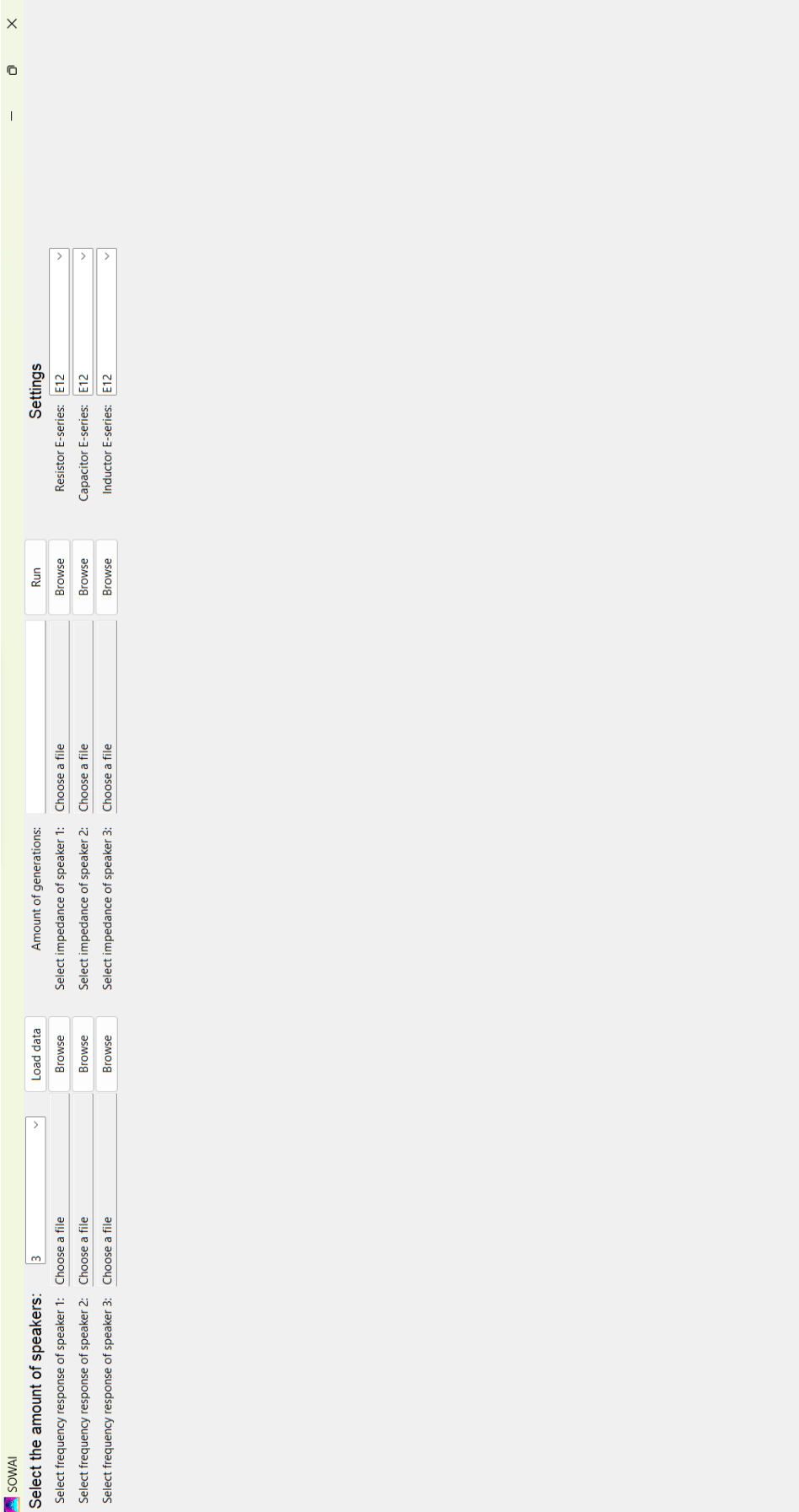
Capactor E-series: E12

⌵

Inductor E-series: E12

⌵

Figure C.1: Initial screen of the GUI



The image shows the initial screen of a GUI application. The window has a title bar with standard OS controls (minimize, maximize, close). The main area is divided into several sections. On the left, there is a logo and the text 'SOWAI'. Below this, the heading 'Select the amount of speakers:' is followed by a dropdown menu set to '3'. Underneath, there are three rows, each with a label ('Select frequency response of speaker 1:', 'Select frequency response of speaker 2:', 'Select frequency response of speaker 3:') and a 'Choose a file' button. In the center, the heading 'Amount of generations:' is followed by a text input field. Below this are three rows, each with a label ('Select impedance of speaker 1:', 'Select impedance of speaker 2:', 'Select impedance of speaker 3:') and a 'Choose a file' button. On the right, there is a 'Run' button. Below the 'Run' button are three 'Browse' buttons. At the bottom right, there is a 'Settings' section with three dropdown menus labeled 'Resistor E-series:', 'Capacitor E-series:', and 'Inductor E-series:', all of which are set to 'E12'.

SOWAI

Select the amount of speakers: 3

Select frequency response of speaker 1: Choose a file

Select frequency response of speaker 2: Choose a file

Select frequency response of speaker 3: Choose a file

Amount of generations:

Select impedance of speaker 1: Choose a file

Select impedance of speaker 2: Choose a file

Select impedance of speaker 3: Choose a file

Run

Browse

Browse

Browse

Settings

Resistor E-series: E12

Capacitor E-series: E12

Inductor E-series: E12

Figure C.2: Initial screen of the GUI

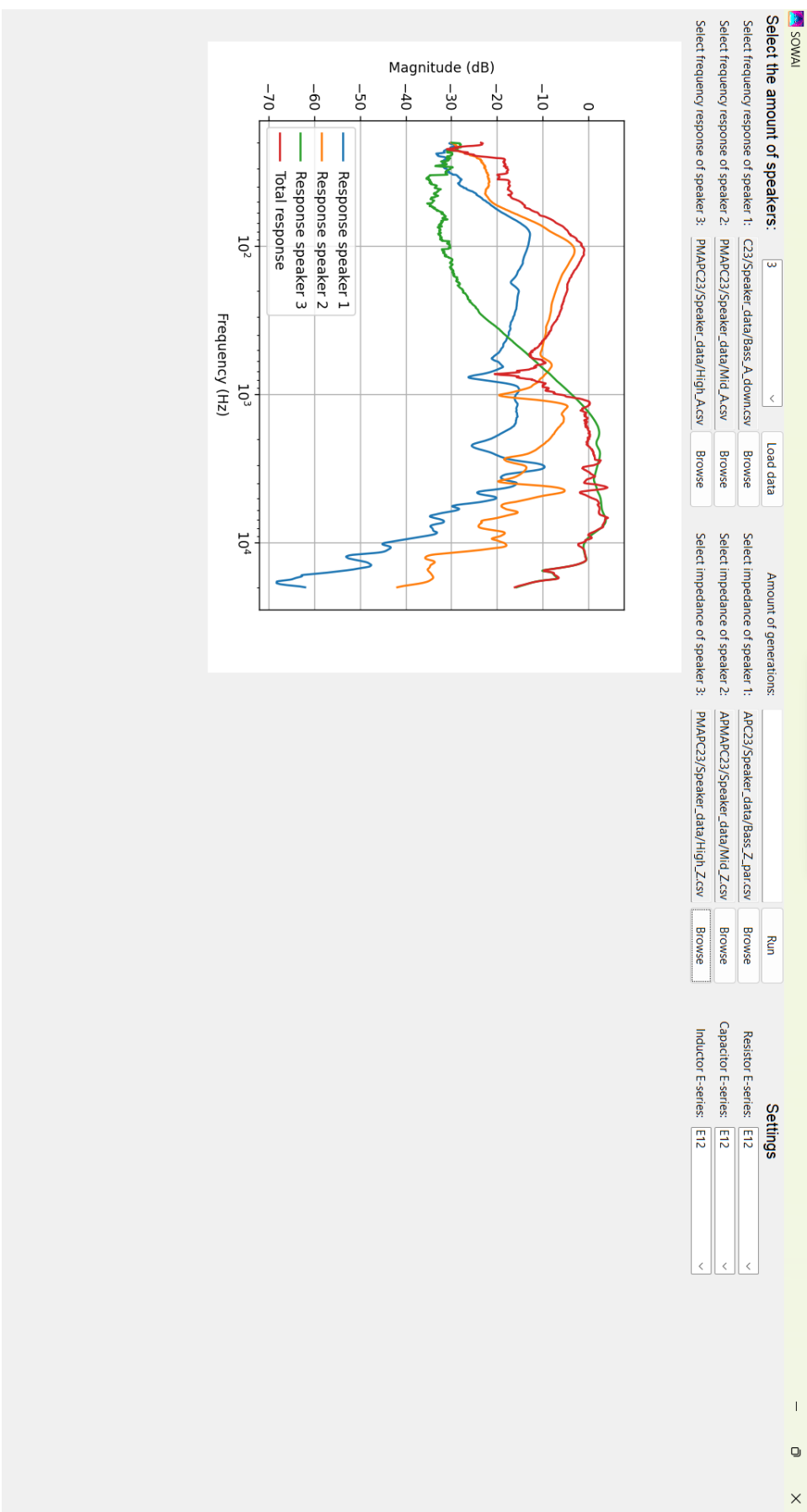


Figure C.3: The GUI after loading data

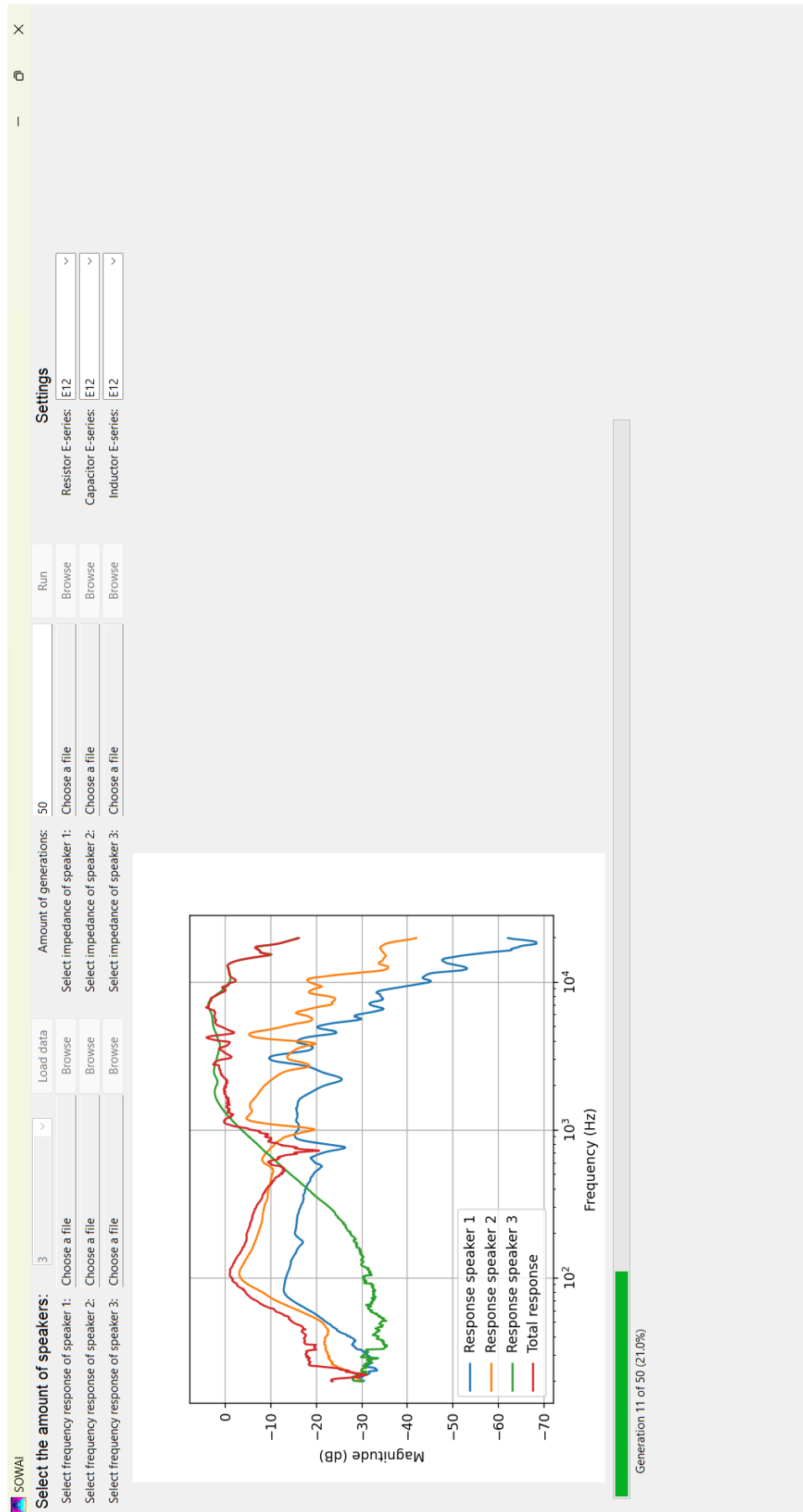


Figure C.4: The GUI during running the main loop

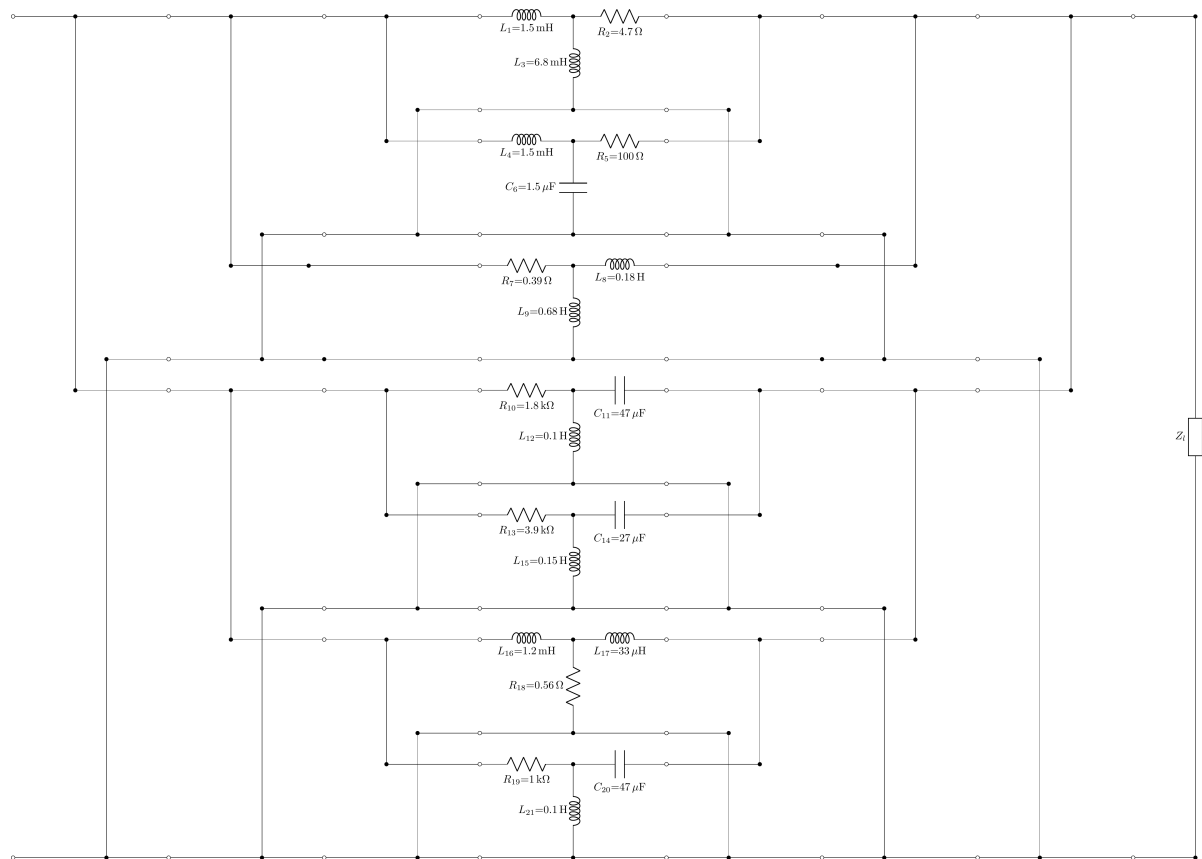


Figure C.5: A generated circuit for the bass speaker

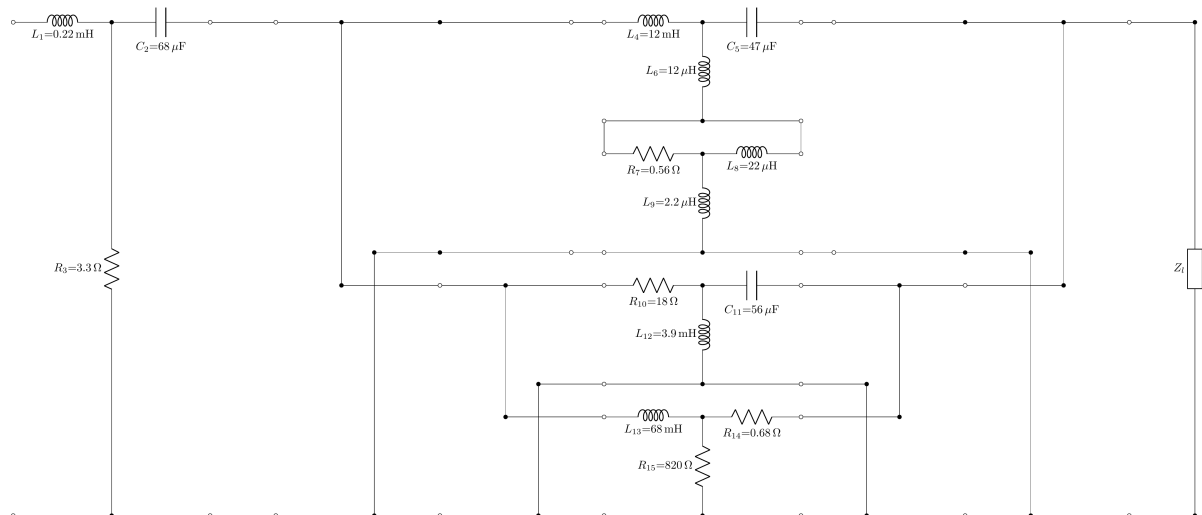


Figure C.6: A generated circuit for the mid-range speaker

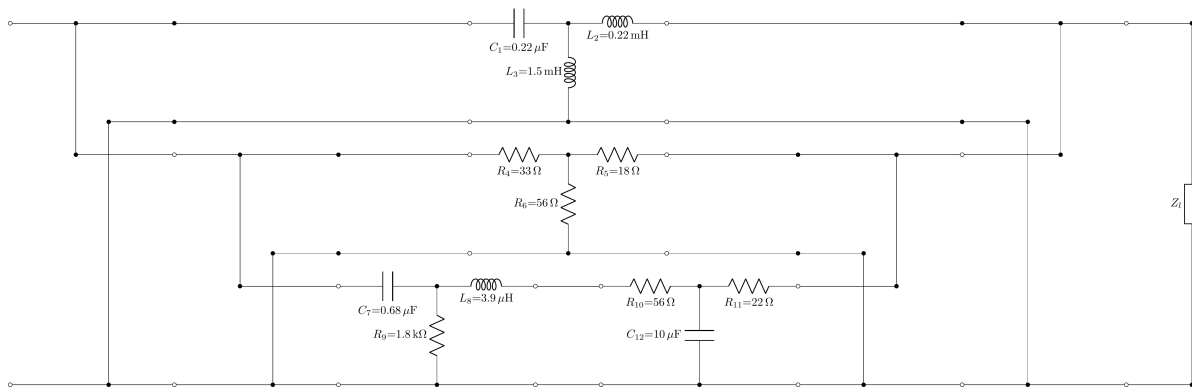


Figure C.7: A generated circuit for the tweeter

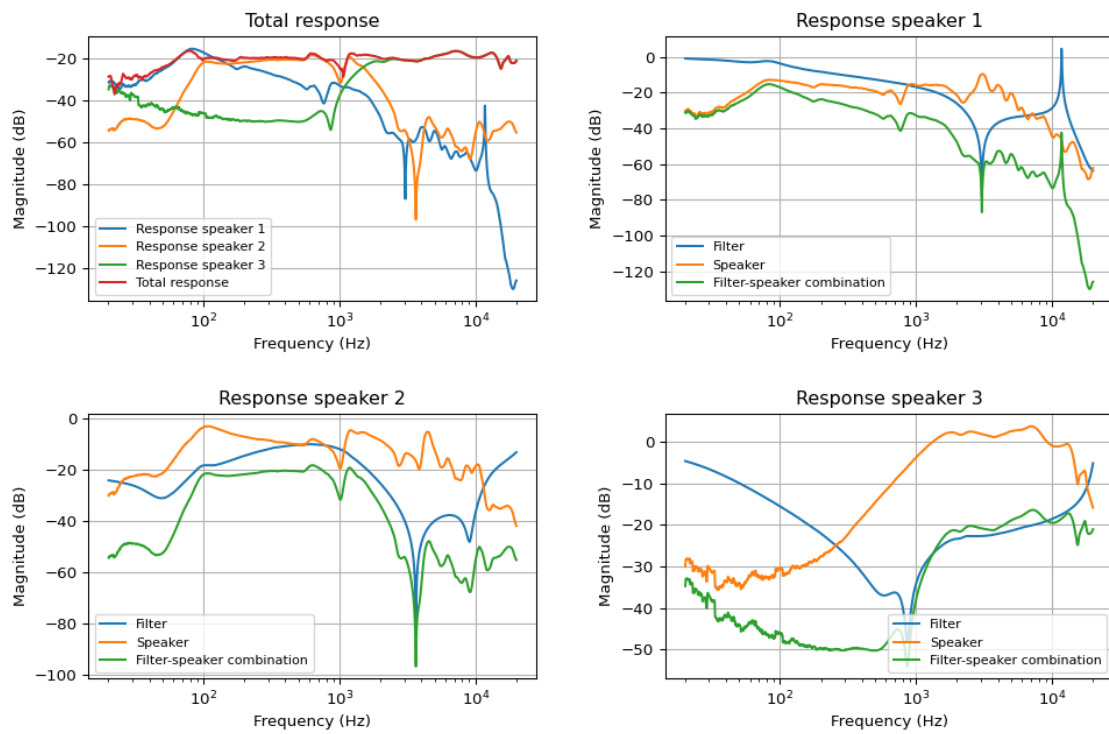


Figure C.8: The final acoustic response of the speaker system if the filters were build

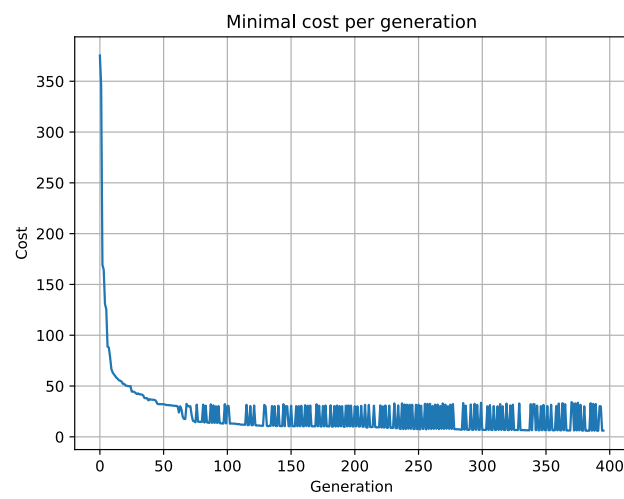
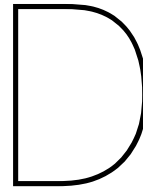


Figure C.9: Minimum cost per generation. The spikes at the end are there, because of the two different costs are used to determine what the best cost is



Best result found

In this appendix, the best results encountered during this project can be found. For this result, the program run for 1823 generations in 3601 seconds and the maximal amount of T-sections per tree was 16 circuits. There were 200 children per population and 70 parents selected from the children with a tournament size of 4 children. Before the removing the components, the cost was 0.8275 and the corresponding are Figures D.1 to D.6. After the optimization, 11 components were removed and the value of 20 components was changed, resulting in a new cost of 0.7923 (4.25% decrease). The figures after the optimization can be found in Figures D.7, D.8, D.10 and D.11. While the differences are minimal, the biggest difference is around 4 kHz. The outlier is reduced from more than 2 dB to around 1.8 dB.

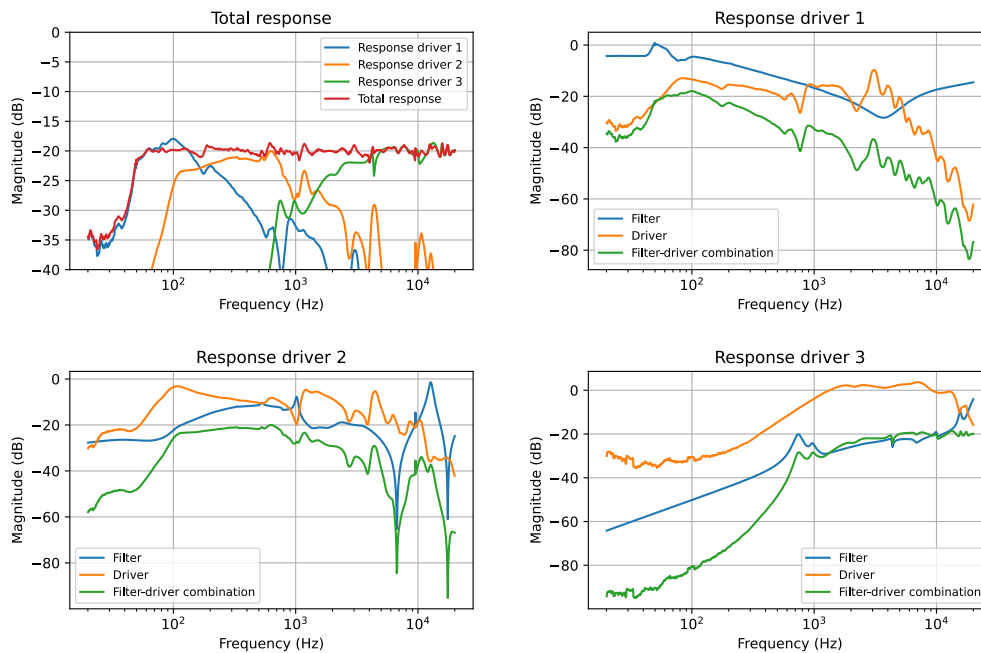


Figure D.1: Frequency response of the drivers and the loudspeaker system

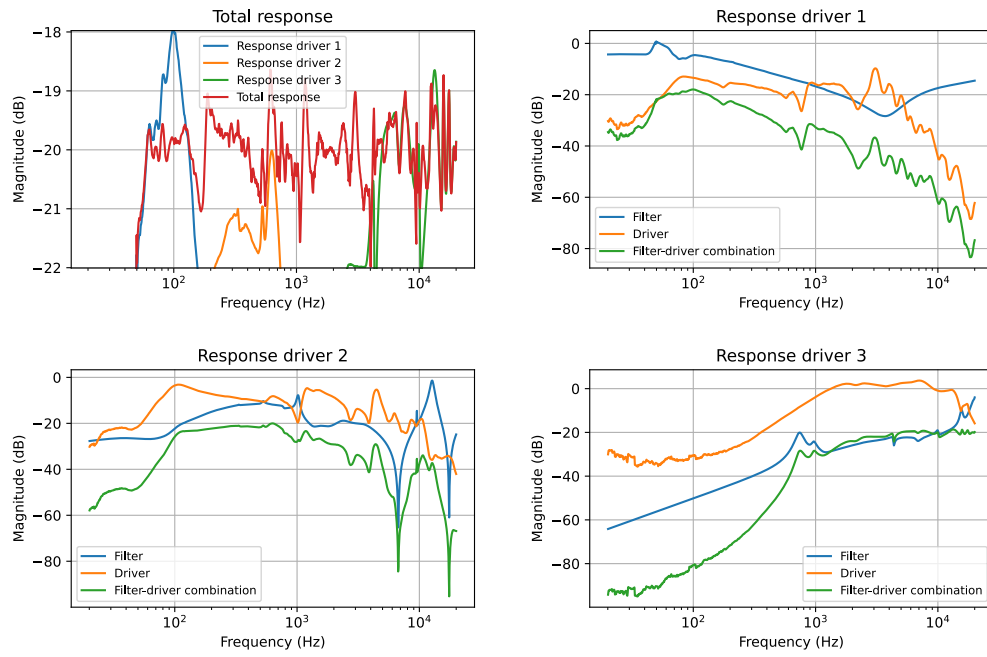


Figure D.2: Frequency response of the drivers and the loudspeaker system, zoomed in to -18 and -22 dB

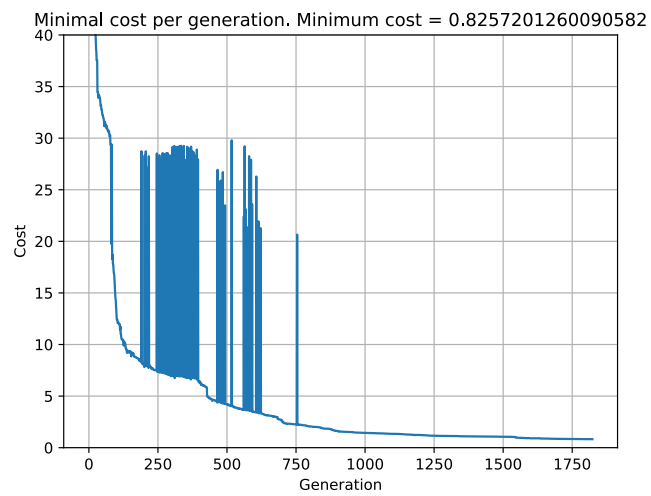


Figure D.3: Lowest cost per generation

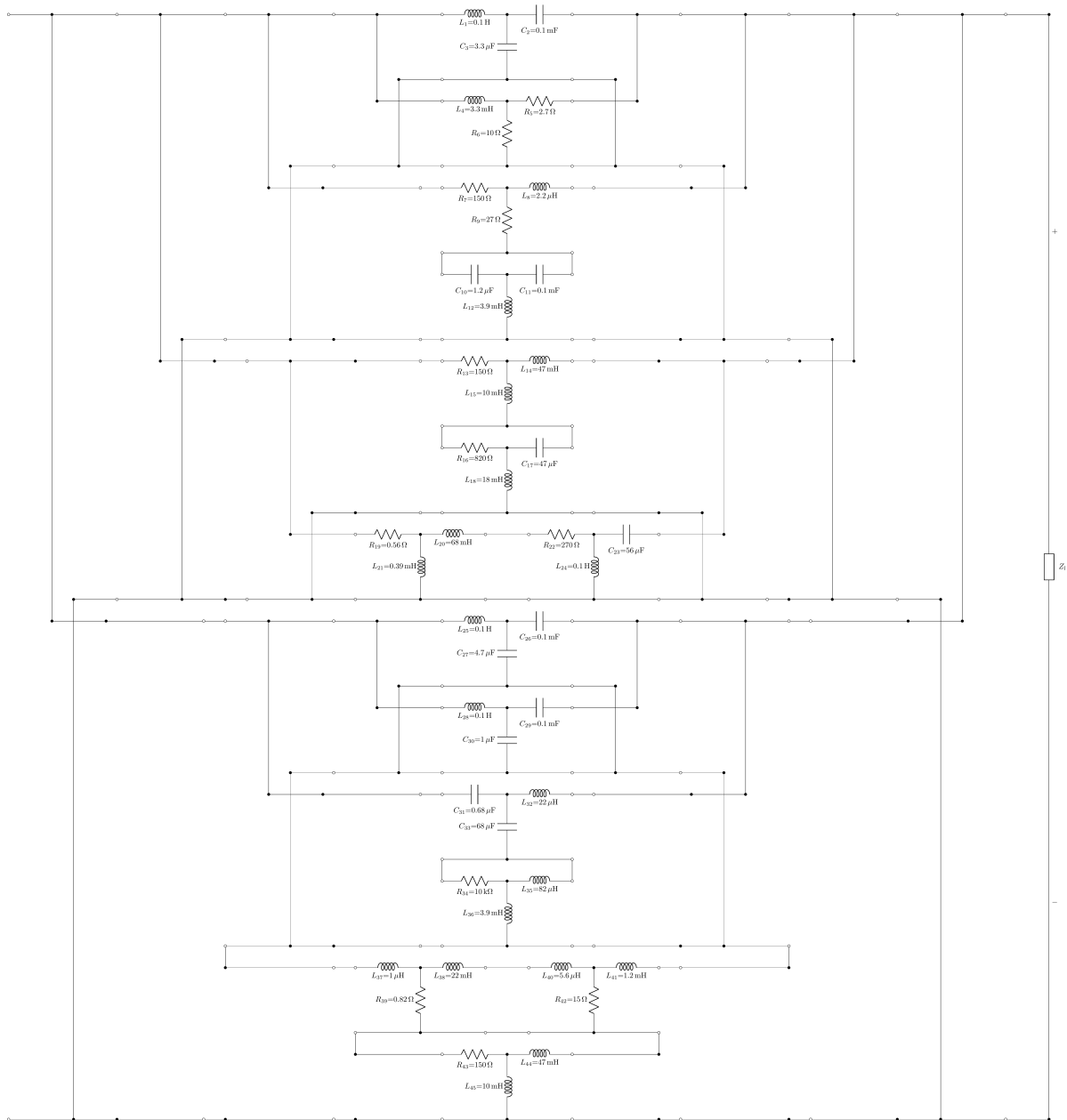


Figure D.4: The circuit for the bass driver

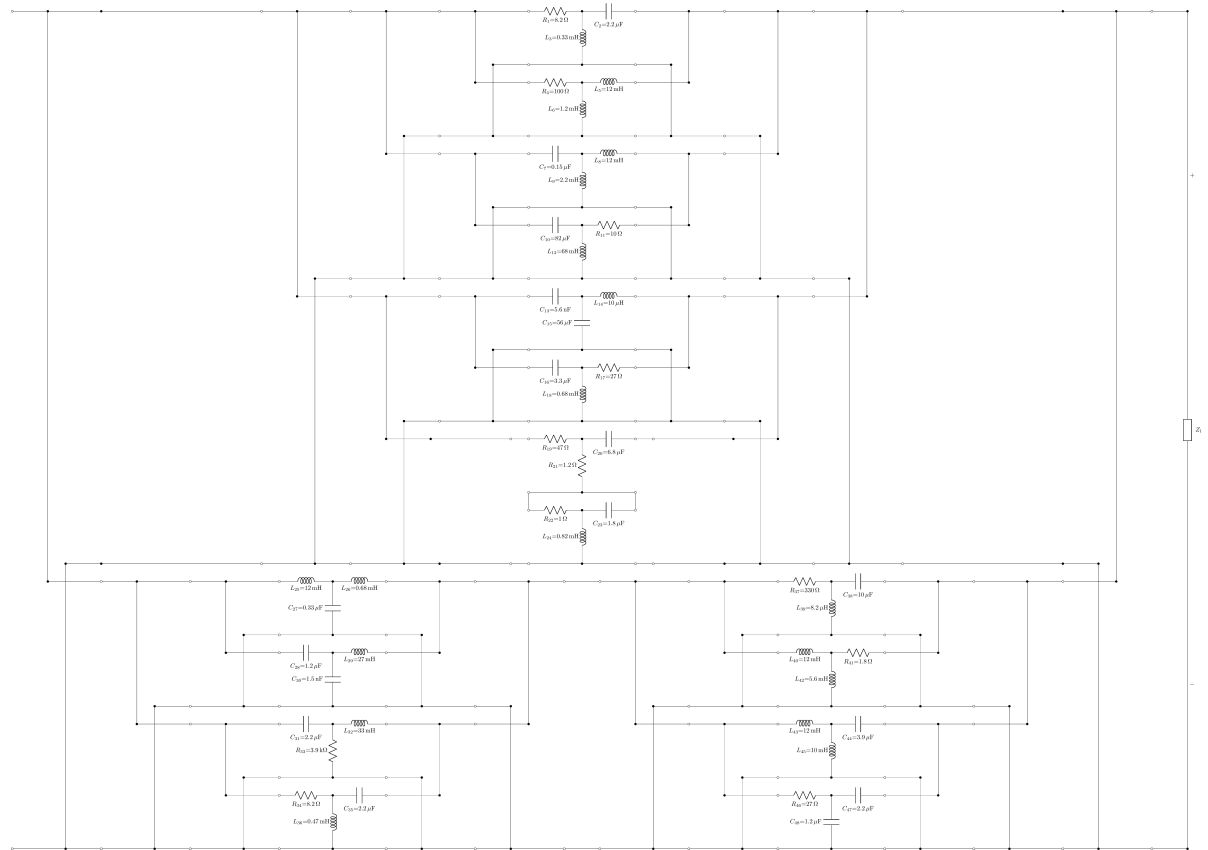


Figure D.5: The circuit for the mid-range driver

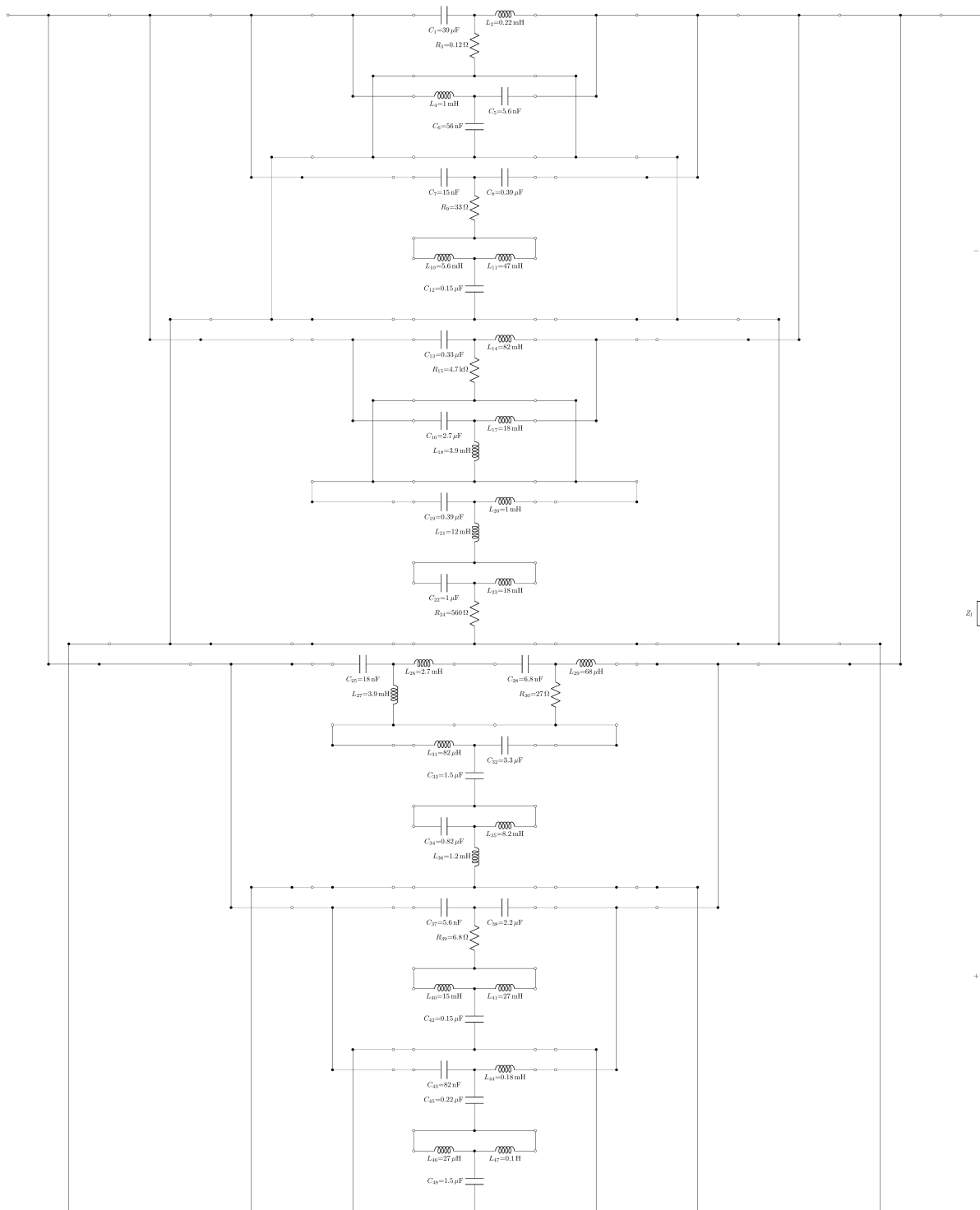


Figure D.6: The circuit for the tweeter

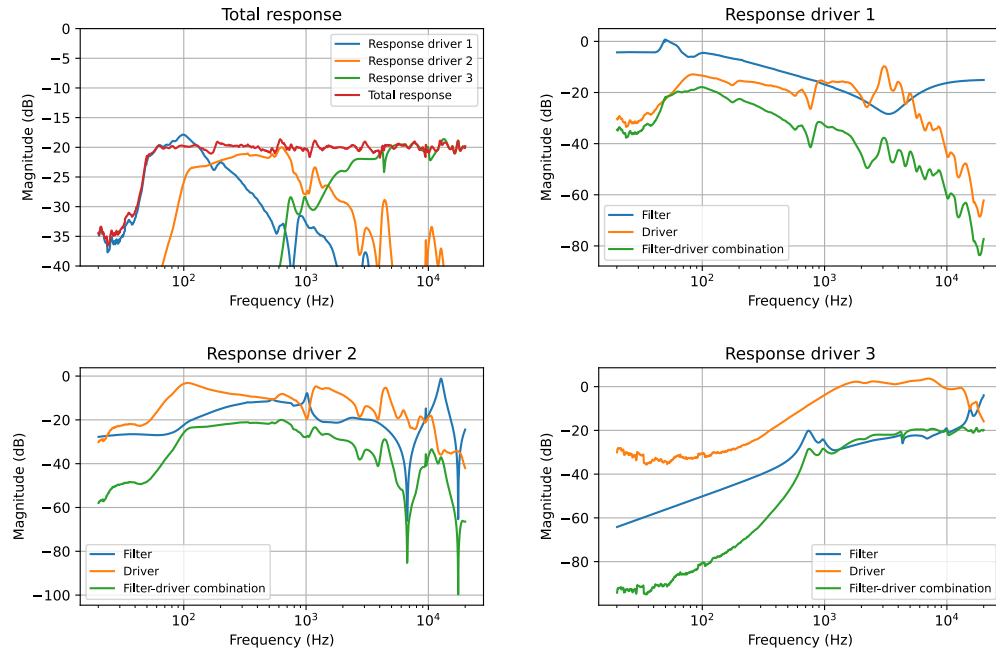


Figure D.7: Frequency response of the drivers and the loudspeaker system after removing components and the last optimization

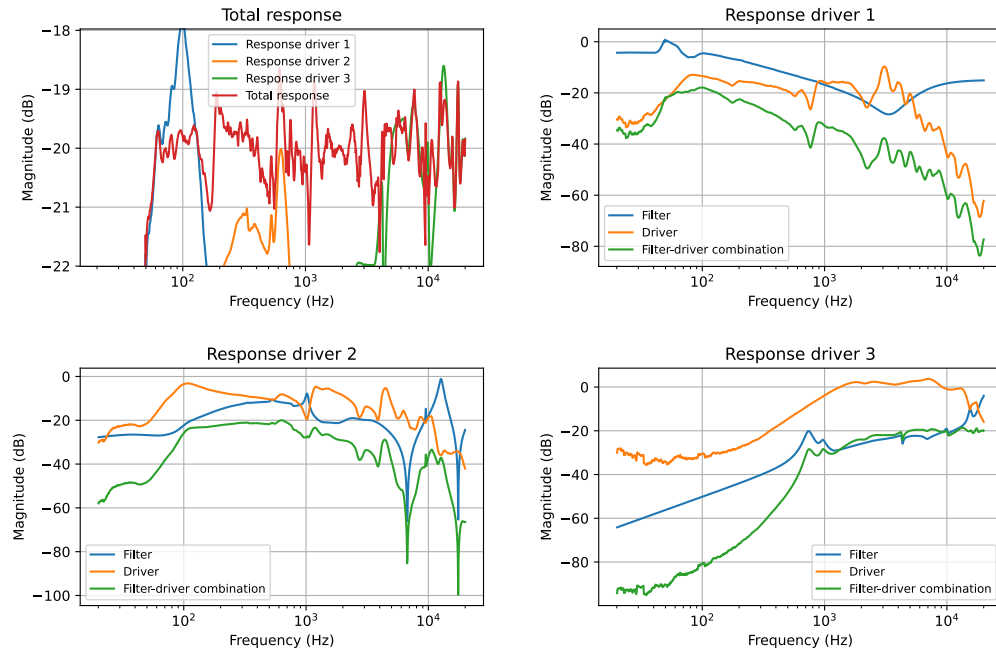


Figure D.8: Frequency response of the drivers and the loudspeaker system, zoomed in to -18 and -22 dB after removing components and the last optimization

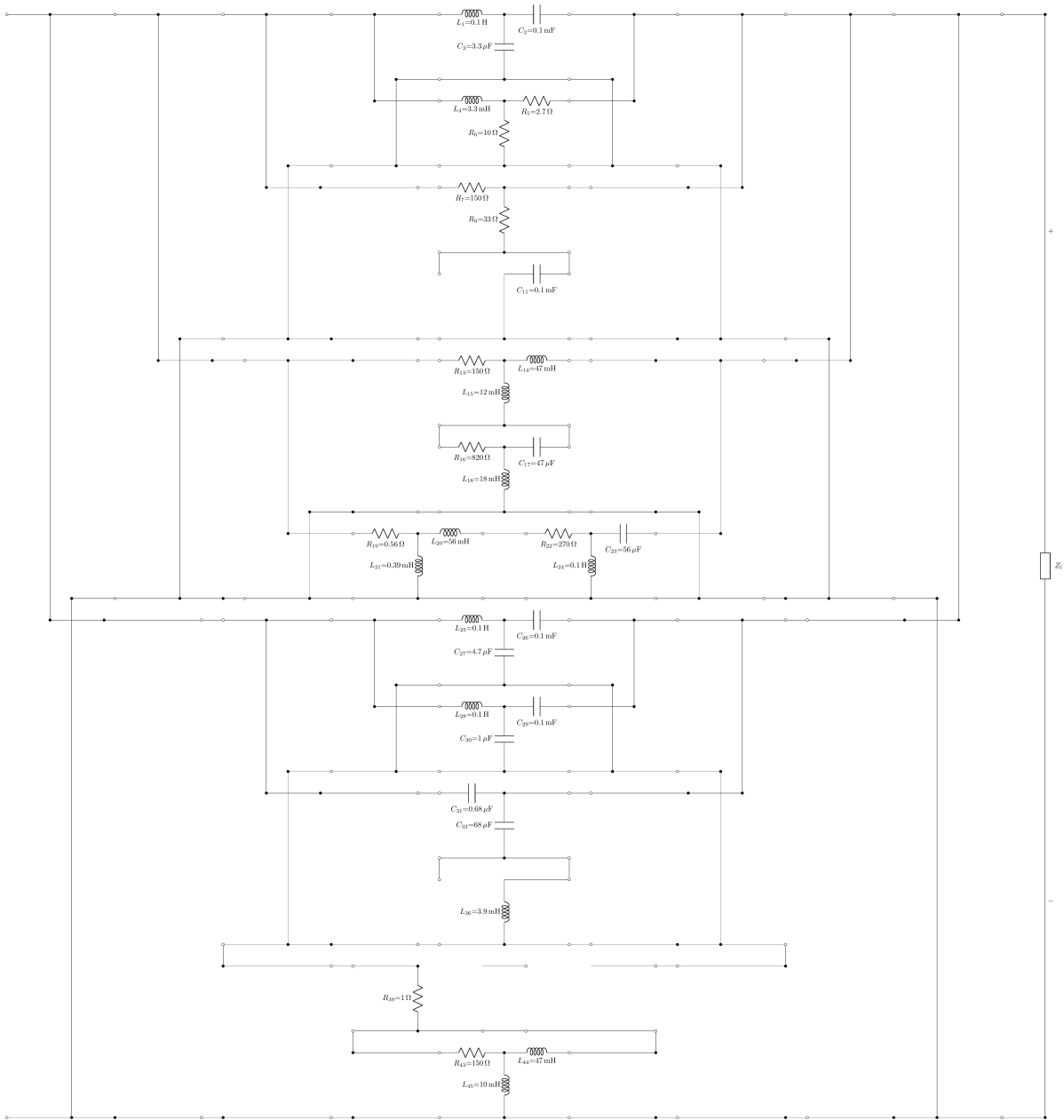


Figure D.9: The circuit for the bass driver after removing components and the last optimization

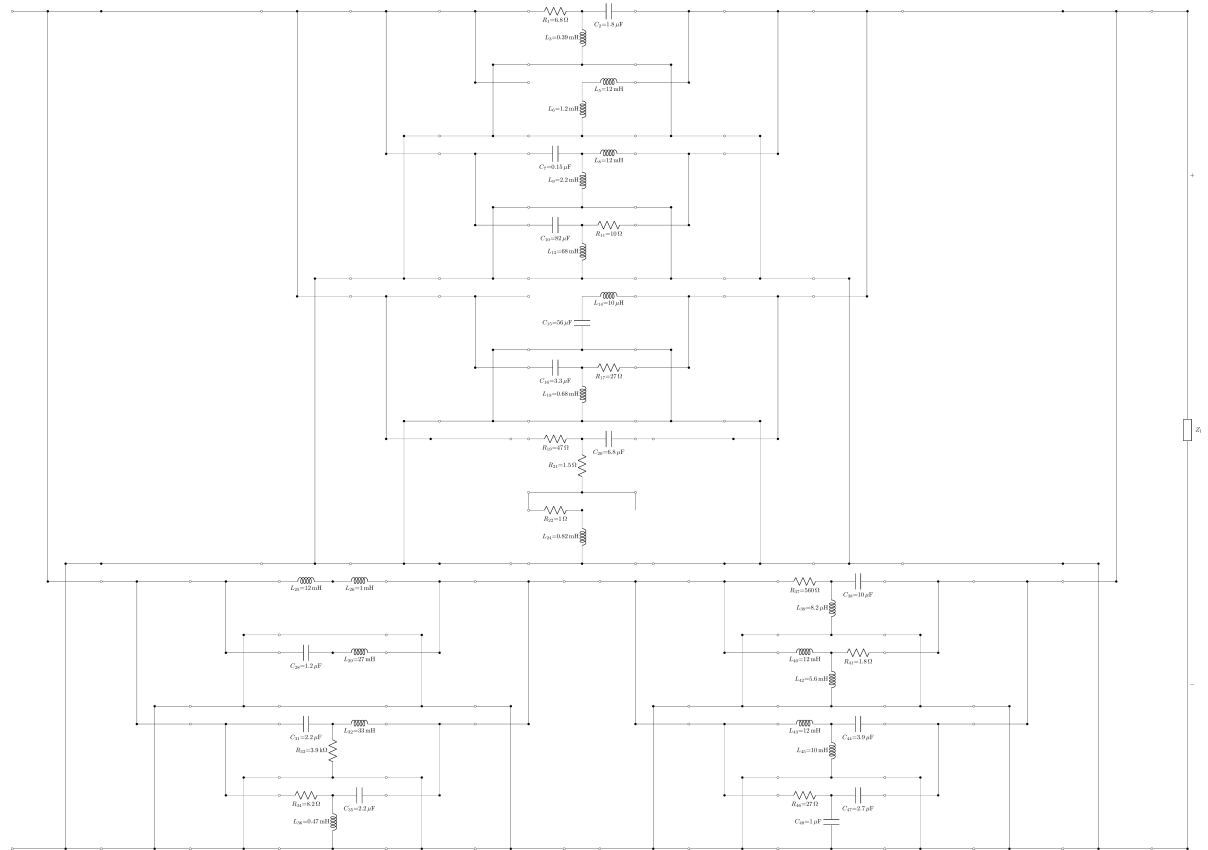


Figure D.10: The circuit for the mid-range driver after removing components and the last optimization

Figure D.11: The circuit for the tweeter after removing components and the last optimization

Bibliography

- [1] Z. Barendse and W. van Overbeeke, "Loudspeaker filter design with AI; electrical circuit representation, mutation, and analysis for AI," Jun. 2023.
- [2] K. Bavelaar and J. Verweij, "Loudspeaker filter design with AI; genetic algorithm selection methods," Jun. 2023.
- [3] T. Golonek and J. Piotr, "Memetic method for passive filters design," in *Analog Circuits*, Y. Wu, Ed., Rijeka: IntechOpen, 2013, ch. 3. DOI: 10.5772/53716. [Online]. Available: <https://doi.org/10.5772/53716>.
- [4] V. Dickason and E. A. J. Bogers, *Luidsprekerkasten ontwerpen*. Segment B.V., 1996.
- [5] O. Aaserud and I. R. Nielsen, "Trends in current analog design—a panel debate," *Analog Integr. Circuits Signal Process.*, vol. 7, no. 1, pp. 5–9, Jan. 1995, ISSN: 0925-1030. DOI: 10.1007/BF01256442. [Online]. Available: <https://doi.org/10.1007/BF01256442>.
- [6] W.-K. Chen, *Passive, Active, and Digital Filters, third Edition* (The Circuits and Filters Handbook, 3rd Edition), 2nd ed. Boca Raton, FL: CRC Press, Jun. 2009.
- [7] H. Zumbahlen, *Linear Circuit Design Handbook*. London, England: Newnes, Feb. 2008.
- [8] S. Särkkä and A. Huovilainen, "Accurate discretization of analog audio filters with application to parametric equalizer design," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 19, pp. 2486–2493, Dec. 2011. DOI: 10.1109/TASL.2011.2144970.
- [9] R. Mina, C. Jabbour, and G. E. Sakr, "A review of machine learning techniques in analog integrated circuit design automation," *Electronics*, vol. 11, no. 3, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11030435.
- [10] R. A. Vural and T. Yildirim, "Component value selection for analog active filter using particle swarm optimization," in *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 1, 2010, pp. 25–28. DOI: 10.1109/ICCAE.2010.5452009.
- [11] N. He, D. Xu, and L. Huang, "The application of particle swarm optimization to passive and hybrid active power filter design," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 8, pp. 2841–2851, 2009. DOI: 10.1109/TIE.2009.2020739.
- [12] X. Han, Y. Liang, Z. Li, et al., "An efficient genetic algorithm for optimization problems with time-consuming fitness evaluation," *International Journal of Computational Methods*, vol. 12, no. 01, p. 1350106, Jan. 2015. DOI: 10.1142/s0219876213501065.
- [13] X. Yan, W. Li, Y. Zhang, H. Zhang, and J. Wu, "Electronic circuit automatic design based on genetic algorithms," *Procedia Engineering*, vol. 15, pp. 2948–2954, 2011. DOI: 10.1016/j.proeng.2011.08.555.
- [14] P. Coelho, J. M. do Amaral, E. N. D. Rocha, and M. Bentes, "Audio circuits evolution through genetic algorithms," in *Proceedings of the 24th International Conference on Enterprise Information Systems - Volume 2: ICEIS*, INSTICC, SciTePress, 2022, pp. 514–520, ISBN: 978-989-758-569-2. DOI: 10.5220/0011062500003179.
- [15] M. Jiang, Z. Yang, and Z. Gan, "Optimal components selection for analog active filters using clonal selection algorithms," in *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues: Third International Conference on Intelligent Computing, ICIC 2007 Qingdao, China, August 21-24, 2007 Proceedings 3*, Springer, 2007, pp. 950–959.
- [16] H. Lan and J. He, "Increasing-dimension evolution: Make the evolutionary design of passive filters more efficient," *Applied Soft Computing*, vol. 131, p. 109740, 2022, ISSN: 1568-4946. DOI: 10.1016/j.asoc.2022.109740.

- [17] J. Pillans, "Efficiency of evolutionary search for analog filter synthesis," *Expert Systems with Applications*, vol. 168, p. 114267, 2021, ISSN: 0957-4174. DOI: <https://doi.org/10.1016/j.eswa.2020.114267>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0957417420309787>.
- [18] C. Goh and Y. Li, "Ga automated design and synthesis of analog circuits with practical constraints," in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, vol. 1, 2001, 170–177 vol. 1. DOI: 10.1109/CEC.2001.934386.
- [19] J. He and J. Yin, "A practical evolution model for filter automatic design," in *2018 14th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, 2018, pp. 406–411. DOI: 10.1109/FSKD.2018.8686881.
- [20] J. He, P. Xia, and W. He, "A novel circuit coding method for circuit evolutionary design," in *2018 14th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, 2018, pp. 400–405. DOI: 10.1109/FSKD.2018.8686984.
- [21] F. Castejón and E. J. Carmona, "Automatic design of analog electronic circuits using grammatical evolution," *Applied Soft Computing*, vol. 62, pp. 1003–1018, 2018.
- [22] Ž. Rojec, J. Olenšek, and I. Fajfar, "Analog circuit topology representation for automated synthesis and optimization," *Electronic Components and Materials*, vol. 48, no. 1, pp. 29–40, 2018.
- [23] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697. DOI: 10.5555/1593511.
- [24] SymPy Development Team, *Lambdify*, May 2023. [Online]. Available: <https://docs.sympy.org/latest/modules/utilities/lambdify.html>.
- [25] Python Software Foundation, *Threading - thread-based parallelism*, Jun. 2023. [Online]. Available: <https://docs.python.org/3/library/threading.html>.
- [26] K. J. Wong, *Multithreading and multiprocessing in 10 minutes*, May 2023. [Online]. Available: <https://towardsdatascience.com/multithreading-and-multiprocessing-in-10-minutes-20d9b3c6a867>.
- [27] A. Ajitsaria, *What is the python global interpreter lock (GIL)?* May 2021. [Online]. Available: <https://realpython.com/python-gil/>.
- [28] Python Software Foundation, *Concurrent.futures - launching parallel tasks*, Jun. 2023. [Online]. Available: <https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ProcessPoolExecutor>.
- [29] M. Hayes, "Lcapy: Symbolic linear circuit analysis with Python," *PeerJ Computer Science*, e875, Feb. 2022, ISSN: 2376-5992. DOI: 10.7717/peerj-cs.875. [Online]. Available: <https://doi.org/10.7717/peerj-cs.875>.
- [30] M. Fitzpatrick, *PyQt vs. tkinter: Which should you choose for your next python GUI?* Apr. 2023. [Online]. Available: <https://www.pythonguis.com/faq/pyqt-vs-tkinter/#tkinter-vs-pyqt-a-feature-comparison>.