Loudspeaker filter optimization with AI

Electrical circuit representation, mutation and analysis for AI

by

Zoyla Barendse & Willem van Overbeeke

a part to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Friday June 30, 2023.

AI generated image [1]

Student number:	5363950 (Willem van Overbeeke)	
Student number:	5373891 (Zoyla Barendse)	
Project duration:	April 24, 2023 – June 30, 2023	
Thesis committee:	Dr. Ir. G.J.M. Janssen	TU Delft, supervisor
	Dr. S. D. Cotofana	TU Delft, jury chair
	Dr. G. Joseph.	TU Delft, jury member

TU Delft

Abstract

This paper reports the design of a part of a genetic algorithm, which is made to design analog filters for loudspeakers. The part in this report is the part which deals with the representation of electrical filter circuits, the mutation of these filters, and finding their transfer function. The considered representations are graph coding [2] and a tree data structure [3]. They are compared on intuitiveness, how well mutations can be performed, and the complexity of calculating the transfer function. The tree structure is reasoned to be the most suitable. Described is which mutations can be performed on a filter by the final program, as well as how embryo circuits are made, how the transfer function is calculated, and which hyperparameters were designed and how they were set. Finally, the design is implemented using Python and the operations are tested.

Preface

This is the final project to complete the Bachelor Electrical Engineering at Delft University of Technology. The goal of the project was to make one step in the design of a sound system easier, namely the design of audio filters for each speaker, using artificial intelligence. The project was proposed by dr. ir. Gerard Janssen, who also was our supervisor for the project.

During this project, we gratefully received the help and advice of our supervisor dr. ir. Gerard Janssen, for which we would like to thank him. We also thank dr. ir. Justin Dauwels for his feedback half-way through, and we thank dr. Geethu Joseph and dr. Sorin Cotofana in advance for being the jury.

And lastly, this subgroup was in constant collaboration with the other subgroups: Jeroen Verweij and Koen Bavelaar, who worked on evaluating the fitness of a filter, and Timo Zunderman and Jonathan Nijenhuis who made the infrastructure, GUI and cost function. We would like to thank them for their advice, help, and 'gezelligheid'.

Willem van Overbeeke & Zoyla Barendse
Delft, Juni 2023

Contents

Glossary

branch A branch is either the right or the left half that comes below a node in a tree.

child A child consists of a filter for each driver. A child is formed by mutating or reproducing a parent.

driver A driver is one of the sound-producing cones in a loudspeaker system.

embryo An initial subcircuit that makes sure the filter is a valid circuit and stays a valid circuit after mutations.

flag A Boolean signal to signify special occasions.

generation One generation is one cycle of mutation and evaluation.

hyperparameter A fixed parameter that influences the behavior of the program.

leaf A leaf is an end in the data structure tree.

node A node is a connection between two branches in a tree.

parent A parent consists of a filter for each driver. The children that pass the selection (done by the Evaluation subprogram), become parents.

tree A tree is a data structure that represents a filter circuit.

weight A weight is the probability weight. It is used to determine the probability that something is chosen

$$P_i = \frac{w_i}{\sum w}$$

where P_i = the probability of result i

w_i = the weight for i

w = the weight

.

1 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 the 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 [2], [18]–[21]. 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 [22]. 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, as it is often used for this purpose in literature, to design the combination of analog circuits [2], [12]–[14], [18]–[20].

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.

The theses of the subgroup Controller and Evaluation can be found in [23] and [24], respectively.

1.4. Analysis Mutator

A suitable data structure is often the trade-off between runtime and the number of possible final filters that can be made. Solutions in the literature have their own way and own data structure to solve the problem. However, they also have some similarities and they can be divided into three groups:

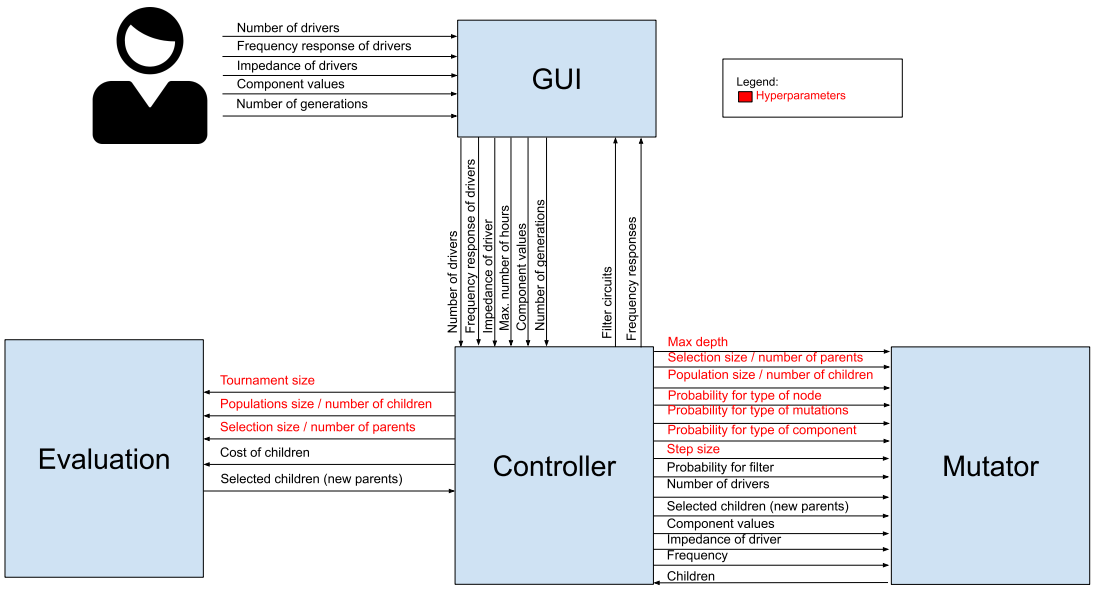


Figure 1.1: A diagram with the subdivision and the signals between the groups

Unlimited possible final filters, limited possible final filters and very limited possible final filters (ladder format).

Methods that produce final filters that are not limited use data structure like graph coding [25], genes structure [18], and a chromosome of 32 bits [26]. The disadvantage of these techniques is that they use SPICE to evaluate the filters or in the case of [26] can only use three components to be able to calculate the transfer function.

In the literature, there are also some examples of methods that try to avoid the use of SPICE and therefore limit the search space. Examples of methods with limited search space are [16], [19], [27]. They cascade predefined building blocks. Therefore, only a ladder structure filter can be an output of these methods.

Two more examples of methods with limited search space are [28] and [3], where [3] is less limited than [28]. They are using a type of tree data structure that gives them the possibility to calculate the transfer functions without the use of SPICE. The possibility of creating creative filters is more limited compared to the listed methods that use SPICE but less limited than the building block methods.

The possible mutations that can be applied to filters depend on the data structure that represents the filter. The mutations that are used in every method are: insert component(s), change value component(s), change type component(s) and crossover. [18] also have a mutation that interchanges the input and the output and [3], [18] have the option to remove components.

1.5. Problem definition Mutator

The goal of the Mutator is to have a way to represent filters, do mutations on them, and calculate their transfer function.

1.6. Thesis synopsis

In this thesis, the program of requirements for the entire system and for the mutator specifically is discussed in Chapter 2. Then in Chapter 3, two different data structures are compared and the design choices are described. In Chapter 4, the implementation of the final design is explained and its functioning validated. A discussion of the results and the conclusion of the project is given in Chapter 5 as well as recommendations for future work.

2

Program of Requirements

2.1. Program of Requirements

The product of this project is a program that generates a filter circuit for the given acoustic transfer and impedance of a speaker system. The requirements for this generated circuit are stated in the program of requirements, below.

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.

2.1.1.1. 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 as specified in Section 2.1.1.2.
4. The program must be able to design a passive analog filter for each driver with a minimal performance as specified in Section 2.1.1.2.
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.

2.1.1.2. Filter constraint requirements

6. The program must determine an operating range of each of the drivers, using the constraints from Section 2.1.1.1, Item 3.
7. The program must be able to design filters which only contain E12 series resistors (1Ω - 1MΩ), capacitors (100pF - 100µF) and inductors (1µH - 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 2.1.1.2.

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.

2.1.2.1. Program requirements

1. The program should be able to design analog circuits for speakers systems varying from two to four drivers.
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.

2.1.2.2. 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.

2.1.2.3. 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. Mutator requirements

The Mutator is a subprogram that builds an initial population (parents), and applies changes, partly random, to create a new set of solutions (children) which will then be evaluated by the evaluation subprogram. The evaluation group will then send back the selected children (parents) to mutate and which then sends the children back to evaluation etc. The first two requirements must be completed in week 3, the rest in week 4.

2.2.1. Mandatory requirements

1. The Mutator must be able to make a valid circuit (embryo circuit) for initializing the program.
2. The Mutator must generate new filters (children).
3. The Mutator must take a group of filters (parents) as input.
4. The Mutator must take the hyperparameters as input.
5. The Mutator must take a list of available components as input.
6. The Mutator must be able to add components to a parent to form a child.
7. The Mutator must be able to delete components from a parent to form a child.
8. The Mutator must be able to change the values of components of a parent to form a child.
9. The Mutator must be able to work with discrete component values as given by the controller.
10. The Mutator must calculate the transfer function of the children.
11. The Mutator must give the symbolic transfer function of the children as output.
12. The Mutator must give the values of the components as output.

2.2.2. Trade-off requirements

13. The Mutator should find the optimal action, e.g. mutation, reproduction or cross-over.
14. The Mutator should design a specific embryo circuit for each driver.
15. The Mutator should take flags as input that describe what change could be beneficial for the filter.
16. The probability of each action that can be applied to a parent should be variable.
17. It should be tried to improve the speed of calculating the transfer function of a child.

3

Design process

In this chapter, the design process is described. Two solutions are presented: graph coding and tree structures. They both have an advantage over a third option, cascading building blocks, as can be seen in Table 3.1. Graph coding, as well as tree structures, can both be used to structure a circuit as data for AI. In the following sections, they are compared on their fitness for the program. After consideration, the tree structure was chosen as the design that was implemented in the final program.

Table 3.1: Comparison of different methods

Method	Possible creative solution	Runtime
Graph coding	Yes	High
Tree structure	Limited	Low
Cascading building blocks	No	Low[a]

[a]Expected value based on the simple computation to calculate the transfer function

3.1. Data structure

This section explains the data structures that are used to represent a filter and their advantages and disadvantages.

3.1.1. Graph coding

The first method that was implemented is the graph coding method described in [25]. It consists of a number of nodes, of which some are connected by components. This results in an upper triangle matrix of admittances. Using graph coding and a matrix representation is very intuitive and allows for the circuit to easily be drawn. A filter could very well be saved by using this data type. However, the operations that should be performed on a circuit were not easily implemented using graph coding; more on this in the following sections.

An example of a graph coding representation is shown in Figure 3.2. It shows an embryo circuit (see subsection 3.2.1) and two mutations (see subsection 3.3.1).

An example of a matrix is given in Equation 3.1 and the corresponding circuit can be seen in Figure 3.1. A connection between two nodes is represented by an admittance on the row and column of the two nodes (with node with the lowest number as the row). For instance, the impedance Z_5 is between nodes 1 and 4, so on row 1 column 4 there is the admittance Y_5. A node cannot be connected to itself, so on the diagonal don't-care's are placed (x). Only the upper triangle is filled as it accounts for all connections.

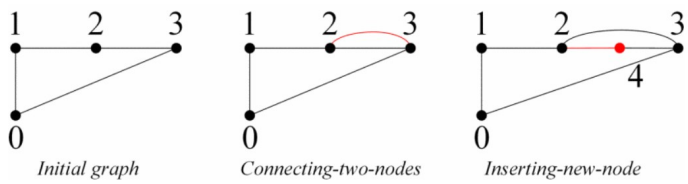


Figure 3.2: Embryo circuit and the two mutations: connecting-two-nodes and inserting-new-node [25]. An edge represents a component, or source or load. A dot represents a node.

$$\begin{bmatrix} x & 0 & 0 & Y_2 & 0 \\ & x & Y_4 & Y_1 & Y_5 \\ & & x & Y_3 & Y_6 \\ & & & x & 0 \\ & & & & x \end{bmatrix} \quad (3.1)$$

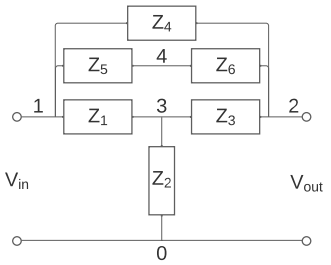


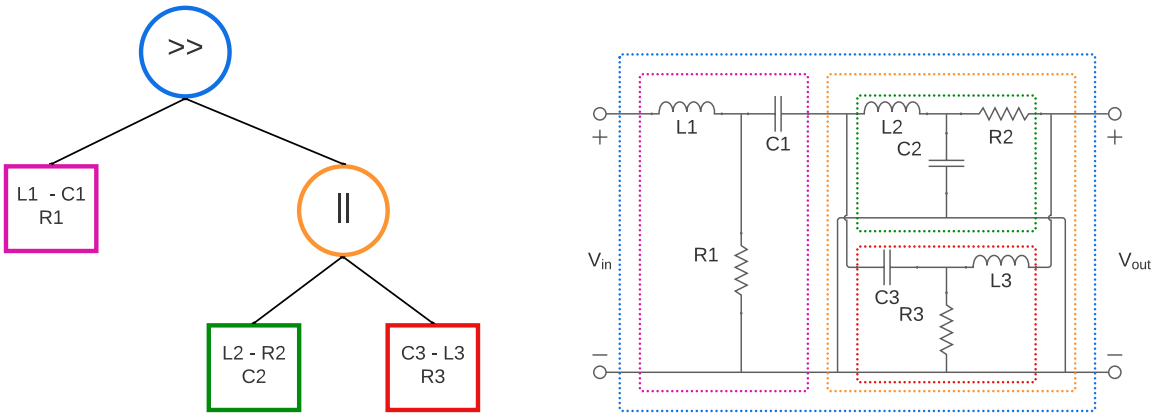
Figure 3.1: The corresponding circuit

3.1.2. Tree

The next circuit representation method is a tree structure as described by [3]. This method uses a tree to represent the filter circuit, which is built from two-ports connected in parallel, in series, or cascaded. Every node, depicted as circles, is a connection of the two two-ports below, resulting in one new two-port. The leaves of the tree are T-structured circuits, which are depicted as squares.

This form of filter circuit representation is not very intuitive. Understanding its functionality is not easy by just looking at the tree structure. A tree structure also limits the number of possible filters that can be created. However, these drawbacks did not outweigh the advantages for calculating the transfer function and the mutations.

An example of a tree structure and its corresponding filter circuit can be seen in Figure 3.3. The T-structured circuits and nodes have been color coded to match their corresponding two-ports.



(a) Tree (b) Circuit

Figure 3.3: A tree structure and its corresponding circuit

3.1.2.1. Nodes

The nodes from the tree represent a connection between two two-ports. The connection can be either series, parallel or cascading. The two two-ports below a node are called the branches.

Cascade The cascade node connects the input ports of the second two-port network with the output ports of the first two-port network, as shown in Figure 3.4.

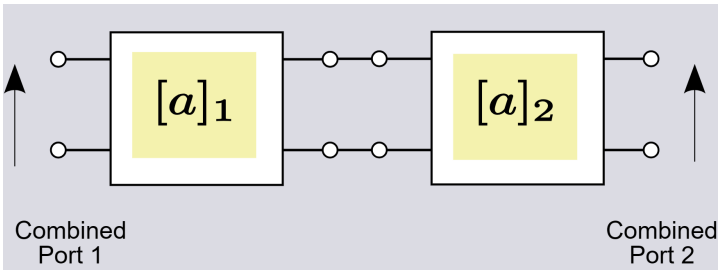


Figure 3.4: Cascade connection [29]

Parallel The parallel node connects the input ports of the two two-port networks in parallel and the output ports of the two two-port networks in parallel, as shown in Figure 3.5.

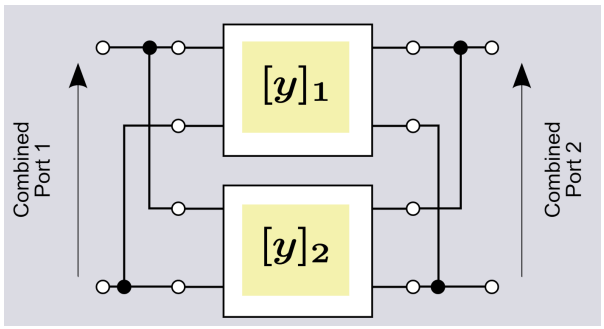


Figure 3.5: Parallel connection [30]

Series The series node connects the negative terminals of the first two-port with the positive terminals of the second two-port, as shown in Figure 3.6. This connection differs from the series connection described in [3]. The reason for this deviation is described in subsubsection 3.4.2.3.

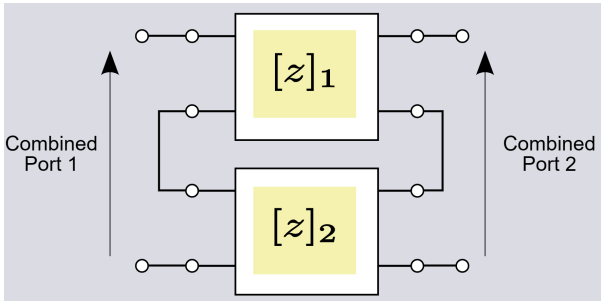


Figure 3.6: Series connection [31]

3.1.2.2. T-circuits

The leaves of the tree are T-structured circuits consisting of three components, see Figure 3.7. Each component can be a resistor, a capacitor, or an inductor. Different from [3], the T-structured circuit is not limited by using each component only once. This deviation was chosen to increase the number of possible circuits that can be created.

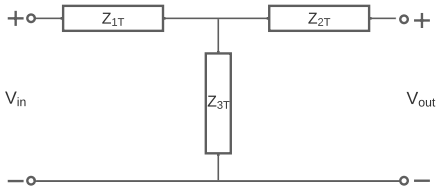


Figure 3.7: T-structured circuit

3.1.2.3. Mutator

The goal of the GA is to create a combination of three filters (or another number, depending on the user for the number of drivers). This is done by the Mutator class. It has a list of all the children and parents, where the children and parents are a combination of three filters. The Mutator also handles the initialization of the population (making embryos) and keeps track of the cost if no changes are applied.

3.2. Embryo Circuits

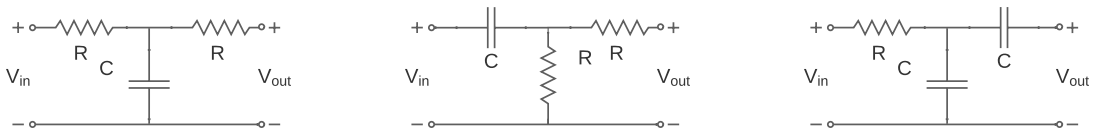
An embryo circuit is an initial circuit that is the start of the algorithm. In [18], the conclusion was made, that using predefined good embryo circuits results in a better final circuit, fewer faulty circuits and lower runtime. This is only the case if the embryo circuit is made with knowledge of how the final circuit should behave. In spite of this, the component values for the embryo circuits are randomly chosen from the component value series. This is done randomly as we do not want to guide the algorithm too much in a common-known solution.

3.2.1. Graph Coding

In the graph coding implementation, the embryo circuits were made by executing a few mutations, but these mutations were not chosen at random but predefined. The embryo circuits were used to make the first generation of a set of valid circuits, by for instance connecting the input and output. The embryo circuit (initial graph) for graph coding can be found in Figure 3.8.

3.2.2. Tree

In the initialization of the tree, the embryo circuits for the tree are built. The embryo circuits are also T-structured circuits. The reason for this is that the tree data structure is kept and all mutations can be applied to the whole filter. If the embryo circuit was not a T-circuit, the mutation would be more difficult as well as the calculation of the transfer function. The transfer function calculation is based on the fact that all leaves are T-circuits as will be explained in Section 3.4.2. Moreover, the advantage of this implementation is that the algorithm still has the possibility to find a better topology itself, by possibly applying mutation on the embryo circuit. Initially, three types of embryo circuits for the tree were designed: low-pass, high-pass and band-pass filters. Each embryo is specific for a driver: the low-pass for a bass, the high-pass for a tweeter, and a band-pass for a mid-range. They all consist only of resistors and capacitors. The choice for using capacitors instead of inductors is because inductors are in general lossier and have higher tolerances than capacitors. The topology of the low-pass, high-pass and band-pass filters is shown in Figure 3.8a, 3.8b and 3.8c, respectively.



(a) Low-pass (b) High-pass (c) Band-pass

Figure 3.8: The embryo circuits

In Section 4.2.3, the impact of the implementation of the embryo circuits is described. The conclusion was made that only the high-pass embryo circuit for the filter for the tweeter is beneficial for the program. Therefore, that embryo circuit is kept and the embryo circuits for the filters for the bass and mid-range drivers are randomly generated T- circuits.

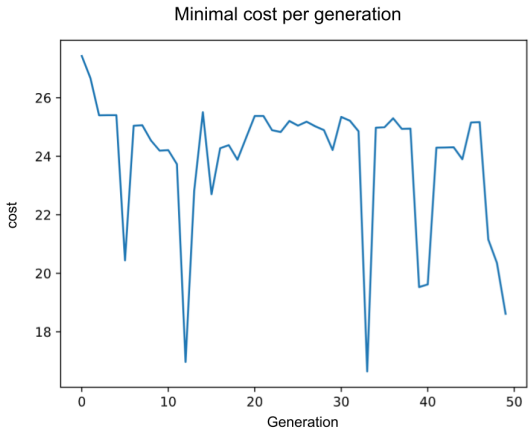


Figure 3.9: The cost fluctuates strongly and does not converge.

3.3. Mutations

In order to create new children from parents mutations need to be applied to the parents. Both graph coding and Tree have multiple options for forming a child from a parent.

3.3.1. Graph coding

Four mutations have been made for graph coding: connecting two nodes, inserting a node, changing a component, and changing the value of a component. A crossover function was not made.

3.3.1.1. Connect node

This mutation chooses two nodes and puts a new component between them, see Figure 3.2. If the two nodes were already connected, the new component is put in parallel.

3.3.1.2. Insert node

Inserting a node is done by choosing two nodes with an existing connection and putting a new component in series, thus creating a new node between the existing component and the new component. The mutation is shown in Figure 3.2. This also results in the matrix getting one more row and column.

3.3.1.3. Component mutations

The last two mutations, change component and change component value, are performed by choosing a random component and either changing the type or the value, respectively.

3.3.1.4. Crossover

Once these four mutations had been implemented, crossover would be the next function to be made. No obvious approach was found and at the same time, problems were encountered trying to get the transfer function. The development of graph coding was therefore halted and thus no implementation was made for crossover with graphs.

3.3.2. Tree

For the tree structure, six mutations have been made: `change component type`, `change component value`, `mutate node`, `add node`, `remove node`, and `tree crossover`. The mutations are listed in the following sections.

When the first basic mutation was implemented, an interesting observation was made. The minimum cost of the population would strongly fluctuate, such as seen in Figure 3.9. This can be interpreted as follows: once a relatively good filter was found, it was quickly lost because the mutations were too harsh. This resulted in some mutations being altered to change less about the filter as well as changes in the Mutator on how mutations are handled.

The Mutator class generates the children from the parents by making a number of copies of each parent and mutating every copy. In an earlier version the Mutator would mutate every one of the three

trees, which was later changed to be only one tree per copy of a parent. This change followed from the fluctuation of the cost, as explained above. With this random choice between the three trees rises the opportunity to give weights to the choice. The weights are given by the controller to signal the Mutator which tree or filter needs some extra mutating.

3.3.2.1. Change component type

The first mutation that was implemented was `change component type`. The mutation takes one component of a T-circuit and changes its type (Resistor, Capacitor, Inductor). Multiple versions have been made to implement this mutation. The first version would change all three of the components, but to make this mutation less severe, this was adopted to only one component. Moreover, firstly the type of component would be chosen randomly. In later versions, the hyperparameter *weights for the type of components* were used to choose the type of component. More about the hyperparameter *weights for the type of components* in Section 3.5.2.

In Figure 3.10 an example is displayed: A capacitor is changed to a resistor (in red).

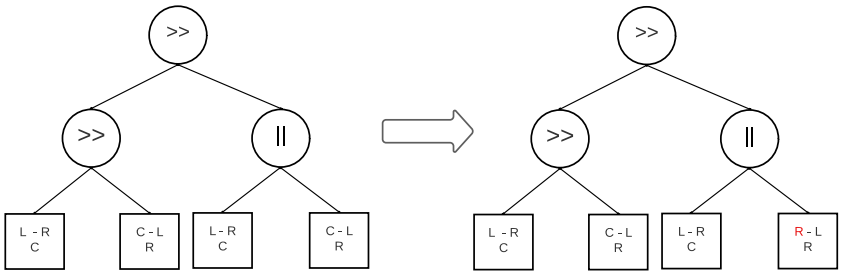


Figure 3.10: The mutation: `change component type`

3.3.2.2. Change component value

The mutation `change component value` gives a new value to one of the components. The new value of the component is randomly chosen from the component value series within a range (hyperparameter: *step size*, see Section 3.5.5) around its previous value, excluding its previous value.

The first implementation of the mutation `change component value` gave a totally random value within the component value series. By using this method, it can take a long time before the optimal value is chosen. Therefore we changed it to picking the component value in ascending order in the component value series. However, this would also take too long, because if its optimal value would be lower than the pre-mutation value, it should go through almost all values in the series. Therefore, the *step size* was implemented, and the new value is chosen within a range, the *step size*, of its previous value. With *step size*, there is a probability that larger steps are taken than one. As it can take larger steps, it can go faster through different values. This will cause it converges faster to its optimum value. It can converge faster because if the new value results in a higher cost than its parent, it has a lower probability of reproducing. This is further explained in the Section 3.5.5.

3.3.2.3. Mutate node

Another obvious mutation that could be implemented, is changing the type of a node. This means that two two-ports will change their connection together. An example can be seen in Figure 3.11, where a parallel connection is changed to a cascading connection (in red). Firstly the new type of node was chosen randomly, in later versions the node was chosen based on the hyperparameters *weights for the type of node*. The chosen weights are explained in Section 3.5.3.

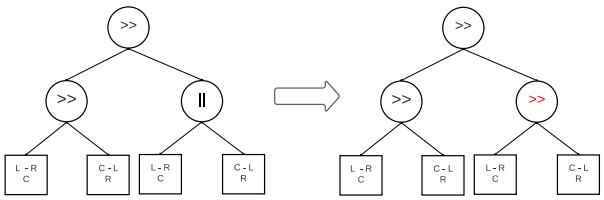


Figure 3.11: The mutation: `mutate node`

3.3.2.4. Mutate from list or Mutate from tree

Some mutations have two implementations. `mutate node` and `change component type` were first implemented as recursive functions. This recursive approach is called `mutate from tree`. For `change component type`, the function starts at the top and keeps choosing random branch until a circuit is found. For `mutate node`, the random choice at each node also includes the node itself, which when chosen means the current node is mutated. This meant that deeper nodes and circuits have a smaller probability of getting mutated.

The other implementation, Mutate from list, chooses randomly from a list of either all nodes or circuits (for `mutate node from list` and `change component type from list`, respectively). This means that every circuit or node has an equal probability of being mutated.

Using Mutate from tree was reasoned to have more impact because deeper circuits have less effect on the transfer function. This type of mutation can be used to improve the diversity of the population, because it changes more of the behavior of a filter with one mutation.

3.3.2.5. Add node

Adding a node is done by selecting a branch of a node and inserting a node. The branch the new node is replacing, will be one branch of the new node. The other branch of the new node is a newly generated circuit. Adding a node can thus also be understood as adding a circuit in parallel, series, or cascaded, to an existing two-port. The type of node was first randomly chosen and later chosen based on the hyperparameter *weights for the type of node*.

In the example in Figure 3.12, the left circuit is replaced by a node (in red) and the circuit is moved to a branch of the new node. The other branch of the new node is a new circuit (also in red).

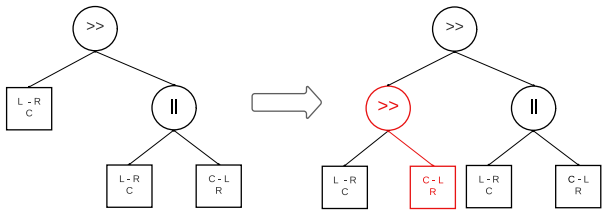


Figure 3.12: The mutation: `add node`

3.3.2.6. Remove node

The mutation `remove node` removes a node from the tree. The place of the removed node is then taken by one of the branches of the removed node. The other branch is lost.

An example can be seen in Figure 3.13, where the right node (crossed out) is removed. One branch takes the place of the removed node and the other is not used anymore.

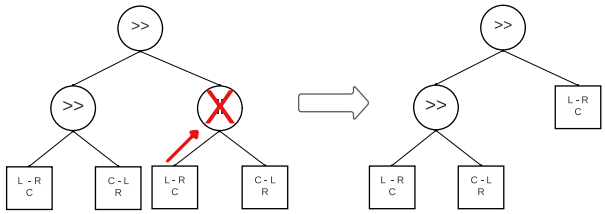


Figure 3.13: The mutation: `remove node`

3.3.2.7. Tree crossover

Crossover with trees is done by choosing one node or circuit in two trees and swapping them. The implementation of this function in the Mutator only uses the first tree, as keeping track of the parents of the second tree is complicated and keeping the second tree would make the code more complicated than necessary. The crossover mutation can thus be understood as replacing a part of one tree with a part from another tree.

A graphical example of the crossover function is shown in Figure 3.14. In the example, a child is created of the blue tree (upper of the two). The black tree (lower of the two) is not changed by the Mutator.

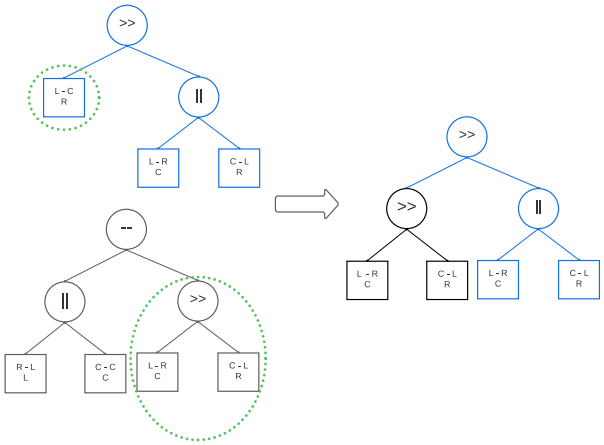


Figure 3.14: The mutation: `tree crossover`

3.3.2.8. Child crossover

This mutation is performed on child level A child consists of one filter (Tree) for each driver in the loudspeaker system. In `child crossover`, one of the filters of the child is replaced by a filter for the same driver from another child. For example, the filter for the bass driver of child two replaces the filter for the bass of child one.

3.4. Transfer function

3.4.1. Graph coding

Calculating the transfer function for graph coding cannot always be done by simply adding impedances in parallel or series. This is due to the fact that complicated filter circuits can be formed using this data structure. Therefore, the transfer function was first calculated using the Python package Lcapy [32]. Lcapy can make a transfer function from a netlist, so the matrix is read out to form the netlist. After implementation, it was found that calculating one single transfer function of a filter with only a few components took already long using Lcapy. An example is shown below, where the transfer function of a relatively simple filter is calculated in 8.69 seconds:

```
Circuit:
 R1 1 3 10
 L1 3 2 0.1
 R2 3 4 3
 C1 4 2 0.01
 L2 4 0 0.001
 R 0 4 3
 Transfer function:
 (3*s**3/16 + 45*s**2/8 + 375*s + 562500)/
 (s**3 + 19695*s**2/8 + 57250*s + 2437500)
 Time it took to calculate the  transfer function:
 8.685295581817627 seconds
```

For the program, the transfer function must be calculated over 10000 times. Thus the conclusion is that it is not possible for the program to use this method to calculate the transfer function for this application.

An alternative to using Lcapy is to calculate the transfer function using nodal analysis. However, after testing if matrix inversion of large matrices was fast enough in Python, the conclusion was that this still takes too long to calculate a single transfer function so this method could also not be used.

3.4.2. Tree

One main advantage of using this tree circuit representation is that the transfer function is easy to calculate. The transfer function is calculated using the same technique as in [3]. This technique calculates the impedances of components of an equivalent T-circuit from two connected T-circuits. The type of

connection determines the calculation. This will be further explained in Section 3.4.2.1, Section 3.4.2.3, and Section 3.4.2.2.

The result is an equivalent T-circuit of the whole tree. The filter is connected to the driver where the impedance of the driver is denoted as Z_L. The resulting equivalent circuit of the filter and driver is shown in Figure 3.15. The transfer function $H(j\omega)$ of this equivalent circuit can be simply derived using circuit theory. The transfer function of the equivalent filter circuit is:

$$H(j\omega) = \frac{Z_L}{(Z_L + Z_2) * (1 + \frac{Z_1}{Z_3}) + Z_1} \tag{3.2}$$

Preferably, the transfer function would be fully symbolic, with the load impedance as well as the frequency and the components as symbols. A fully symbolic transfer function would give the option to evaluate the influence of a single or group of components purely on the transfer function. However, the computation time of a fully symbolic transfer function was too long. The Evaluation subgroup originally needed a partly symbolic transfer function with Z_L and $j\omega$ as symbols, therefore this was implemented. This reduced the runtime to be acceptable. However, in the end, this was no longer needed for the Evaluation subgroup. In the third and final design $H(j\omega)$ is only calculated numerically. From the controller, the impedance and frequency are received and these arrays are used to directly calculate the impedance of the components and thus the transfer function. This results in an array that represents the transfer. This adjustment resulted in a runtime that decreased by a factor of ten, allowing the program to run thousands of generations per hour.

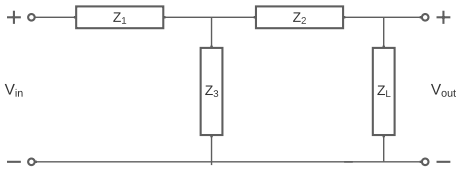


Figure 3.15: Equivalent filter circuit of the tree

3.4.2.1. Cascading (> >)

When cascading two T-circuits, the output of the one T-circuit is connected to the input of the other T-circuit as can be seen in Figure 3.16. An equivalent T-circuit can be made by first adding Z_{a2} and Z_{b1}, then converting the resulting Π circuit, consisting of Z_{a3}, $Z_{a2} + Z_{b1}$ and Z_{b3}, into a T-circuit using the calculation from Appendix A.2.2. Lastly, Z_{a1} and Z_{b2} should be added by their in-series adjacent impedances. The calculations for the conversion from two cascading T-circuits into their equivalent T-circuits are in Appendix A.1.1

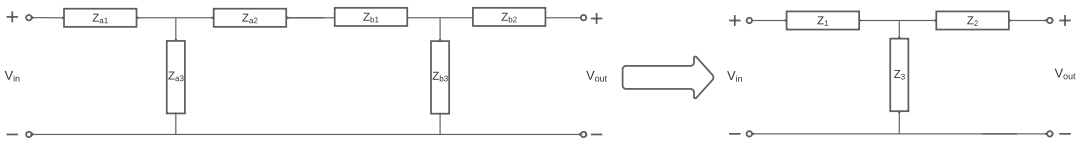


Figure 3.16: Two T-circuits cascaded

3.4.2.2. Parallel (||)

Connecting two T-circuits in parallel results in the circuit of Figure 3.17. In order to obtain the resulting T-structured equivalent circuit, the two T-circuit should be transformed into Π-structured circuits. This is realized using the T to Π transform equations in Appendix A.2.1. This results in two parallel Π-circuits, where impedance Z_{a1} is in parallel with Z_{b1}, Z_{a2} with Z_{b2} and Z_{a3} with Z_{a1}. Adding these impedances in parallel, results in one Π-circuit. Transforming this Π-circuit using the calculation in Appendix A.2.2, will result in the equivalent T-circuit.

The calculations for obtaining an equivalent T-circuit from two T-circuit connected in parallel are shown in Appendix A.1.2.

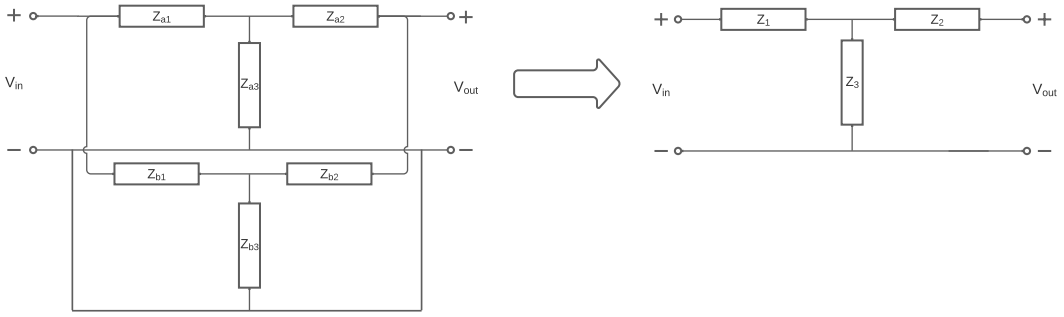


Figure 3.17: Two T-circuits in parallel

3.4.2.3. Series (- -)

Two T-circuits connected in series are shown in Figure 3.18. The equivalent T-circuit is obtained by adding Z_{b1} and Z_{b2} in parallel and adding the resulting impedance with Z_{a3} and Z_{b3} in series to form Z_3. Z_{a1} will become the impedance Z_1 and Z_{a2}, Z_2. The calculation are further explained in Appendix A.1.3.

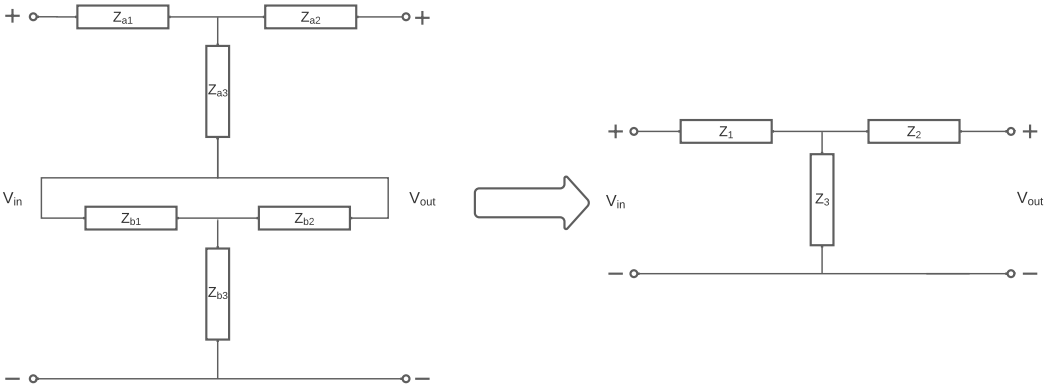


Figure 3.18: Two T-circuits connected in series

At first, the series implementation of [3] was used, which can also be seen in Figure 3.19. The series connection described in the paper connects the two T-circuits as shown in Figure 3.20. The advantage of connecting the two T-circuits as in [3] instead of the common two-port series is that in the equivalent T-series, all three impedances change value. According to [3], the equivalent circuit of this series connection can be calculated by adding in series Z_{a1} with Z_{b1}, Z_{a2} with Z_{b2} and Z_{a3} with Z_{b3}.

Calculations of the equivalent series connections as done in [3]:

$$Z_1 = Z_{a1} + Z_{b1} \tag{3.3}$$
$$Z_2 = Z_{a2} + Z_{b2} \tag{3.4}$$
$$Z_3 = Z_{a3} + Z_{b3} \tag{3.5}$$

The disadvantage of this series implementation was found during testing. It was discovered that the equivalent circuit calculations were not always valid when the series connection was in a larger tree. An example of the difference between the series connection and the way the equivalent circuit of this series connection is calculated can be seen in Figure 3.22. The resulting output of both circuits can be found in Figure 3.21. It is clear that they both give different outputs, which proves that the equivalent circuit calculations are not always valid. This is the reason we used the series connection described above.

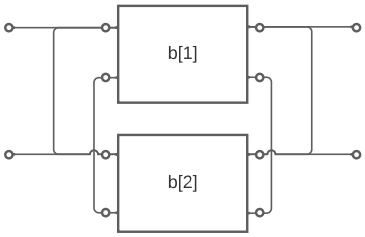


Figure 3.19: Old implementation of series connection

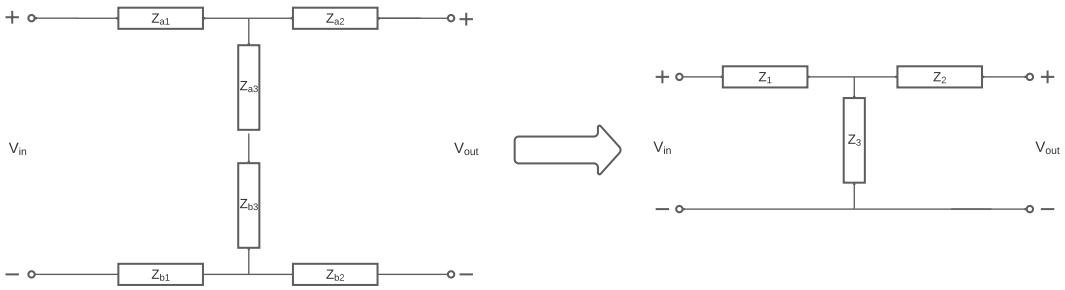


Figure 3.20: Two T-circuits in old-series

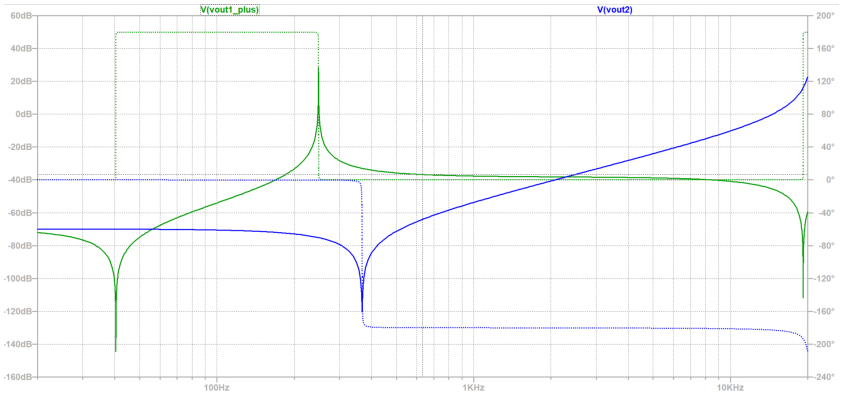
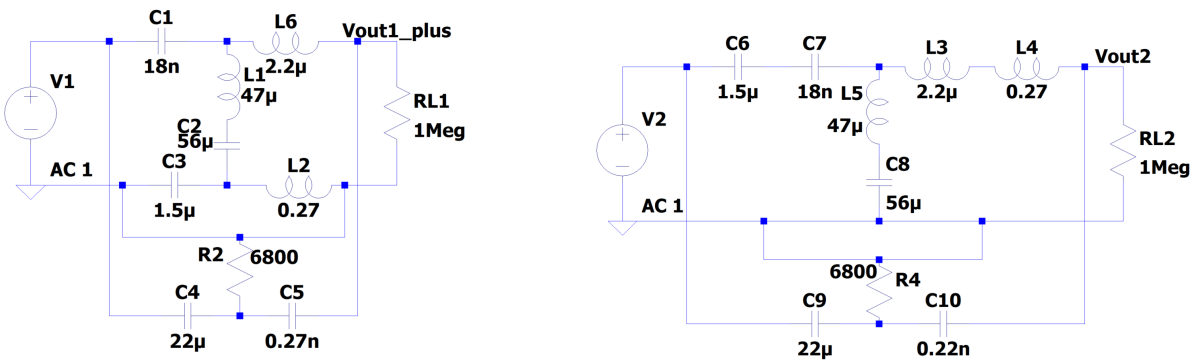


Figure 3.21: Plot of the output voltage of both the circuits from Figure 3.22.



(a) Series

(b) Series implemented as equivalent calculation

Figure 3.22: Circuit with two T-circuits in series in parallel with a third T-circuit

3.5. Hyperparameters

Hyperparameters are input variables of the program that can be tuned to optimize the working of the program. All values for the hyperparameters are fixed during a run. However, for possible future adjustments, it is possible to change each of them during a run. Seven hyperparameters were implemented. They will be discussed in the following subsections.

3.5.1. Max depth

The hyperparameter *max depth* determines the maximum depth of the tree. It, therefore, determines the maximum amount of T-circuits in a final filter, and thus the maximum amount of components in a final filter, see Equations 3.6 and 3.7. Moreover, the *max depth* also influences the runtime of the program. The higher the maximum depth, the more complicated the filters, the more complex the transfer function, and the longer it takes for the program to determine the transfer function.
The value of *max depth* was first set to 4, to minimize the component that could be used. This would result in a maximum of 24 components per filter. However, after some test runs it was found that a *max depth* of 5 would find a lower cost for the three filters faster, see Section 4.2.4.1.

$$Maximum\ number\ of\ circuits\ =\ 2^{(max\ depth-1)} \tag{3.6}$$

$$Maximum\ number\ of\ components\ =\ 3 * Maximum\ number\ of\ circuits \tag{3.7}$$

3.5.2. Weights for the type of component

This hyperparameter gives weights to each component type (resistor, capacitor and inductor). Due to this hyperparameter, when a new circuit (not an embryo) is generated, the probability of choosing one component can be higher than another. To determine the weights, 12 test runs, each of one hour, were done. In the filters of the best child, the components were counted. The results are displayed in Table 3.2. The final weights are chosen by using the ratio between the percentages which are rounded to the nearest ten.

Component type	Count	Percentage of total number of components	Final weights
Resistor	90	21.1%	1
Capacitor	154	36.1%	2
Inductor	182	42.7%	2

Table 3.2: Number of components of each type in 12 runs and the final *weights for the type of component*

3.5.3. Weights for the type of node

The parallel connection was found to appear more often in the final filters than the other two node types do, as can be seen in Table 3.3. In this table, the number of each type of node appearing in the final filter was counted. In total 12 runs were done, each of one hour. For this reason, the probability for the nodes was implemented to help the program converge faster to its optimal filter by giving the parallel connection a higher weight. This way the node type parallel would be chosen more often and the program makes more circuits with parallel nodes, which will improve the speed of the program. *Weights for the type of node* are set by using the ratio between the percentages which are rounded to the nearest ten. This results in the weights shown in Table 3.3.

A downside of having non-equal *weights for the type of node* as well as for the type of component is that the program is guided toward a certain solution. However, as the *population size* is 200 (set by the Evaluation group), each containing 3 filters (assuming a three-way loudspeaker) still in a population around $200 * 3 * 20\% = 120$ series nodes will be present, assuming each filter has a node. Therefore, if a series or cascade node would be more beneficial for a final filter, they will still be passed through during selection (done by the Evaluation group).

3.5.4. Weights for the type of mutations

Each mutation can receive a weight from this hyperparameter. Some mutations have a larger impact on a filter than others, therefore it was thought to be useful to do some mutations more often than others. However, as will be discussed in Section 4.2.4.2, this was not the case. Thus all mutations have equal probability.

Node	Count	Percentage of total number of nodes	Final weights
Series (- -)	27	18.5%	1
Parallel (||)	79	60.8%	3
Cascade (»)	24	20.8%	1

Table 3.3: Number of the types of nodes in 12 runs and the final *weights for the type of node*

The *weights for the type of mutation* was first chosen on trials, then adjusted based on educated guesses, and changed back after more trials. It is difficult to determine the optimum weights for three reasons. The first is that there are lots of possible combinations for the weight and testing takes a lot of time. The second reason is that the perfect weights will differ per cost or per generation, as the next optimum mutation will differ per current filter. The third is that an optimum value is difficult to determine as on the one hand, you want the weights to cause the cost to drop as fast as possible, but on the other hand, you don't want to go into a local minimum. Multiple runs were done to make an attempt to determine the optimum weights. The first runs were done in an early stage of the implementation and the results were discarded as the function that determines the cost did not work properly yet and thus no valid conclusion could be drawn.

The educated guesses are based on some known properties of a Tree. If we take for example a Tree of *max depth* 4, then the Tree has a maximum of 7 nodes, 8 circuits, and 24 components. Both a node and a component can be one of three types. The number of values the components can have depends on the possible component values (E-series) chosen by the user. The most common series is the E12 series. If, for example, a range of 5 decades is chosen, with an E12 series, there are $5*12 = 60$ component values per component type. There can thus be more options to be tested for component values than for components type and even fewer for the node type. With this knowledge the weights were chosen as shown in Table 3.4. However, as stated these are not the final weights chosen.

Table 3.4: Possible values for the hyperparameter: *weights for the type of mutation*

Mutation	Weight
Change component type from list	2
Change component type from tree	1
Mutate node from list	1
Mutate node from tree	1
Remove node	1
Add node	2
Change component value	10
Tree crossover	2

3.5.5. Step size

This hyperparameter influences the mutation: `change component value`. The new value for the component is randomly chosen within this *step size*. A wide *step size* will result in going through different value ranges faster, as it can take bigger steps. However, once it is close to its optimal value it will take longer to get to the optimal value, as it also has a lot of other values in the range it can pick. A narrow *step size*, results in the opposite. The value for the *step size* is chosen at 10. This means that the values chosen can be 1 of the 5 values below its previous value or 1 of the 5 values above its previous value in the component value range. This is thought to be a value that balances the pros and cons of wide and narrow *step sizes*.

3.5.6. Number of parents/selection size

The size of the initial populations and the size of the selected children, who became parents from the evaluation. This hyperparameter is set to 70 by the Evaluation subgroup.

3.5.7. Number of children/population size

The number of children that are formed from parents. This hyperparameter is set to 200 by the Evaluation subgroup

4

Prototype implementation and validation

4.1. Prototype implementation

A child consists of as many trees as the number of drivers. A tree consists of nodes and circuits and a circuit consists of three components. The program, to create and mutate these trees is written in Python, using object-oriented programming. The classes and methods and their connections are shown in Figure 4.1. A hierarchical structure is used to implement the program. On top of the hierarchical structure is the *Mutator* class. The *Mutator* class takes all the hyperparameters, component value series, and the number of drivers as input and initializes the list of parents by calling the class *Tree*. The only method in *Mutator* is `create_children`, this method is called by the subgroup Controller. This method creates the children in two ways. The first is by reproducing and the second is by calling the method `mutate` of one of the *Trees* the initial parents or the parents (selected children) that are chosen by the evaluation. Only after a mutation of a *Tree*, the new transfer function is calculated, by calling the method `get_transfer` of the class *Tree*. Because only one *Tree* of a child is changed at a time, it saves run time to only calculate the transfer function when a child is mutated. Another function of the Mutator class is that it saves the cost of a child and this cost is only amended when the child is changed. On top of this, the *Mutator* class builds a list of possible trees to do the crossover with that is, together with the hyperparameter *weights for the type of mutations*, passed on when the method `mutate` is called.

The class *Tree* receives the hyperparameters *max depth*, *weights for the type of component*, *weights for the type of node*, and *step size* and the number of drivers. Examples of attributes of the class are a list of the circuits, a list of nodes, depth, and the transfer function. When the class is called the embryo circuit is made, by calling the class *GenotypeCircuit*. The method `mutate`, makes a list of mutations (other methods of *Tree*) that are possible on this specific *Tree* and chooses one of them based on the hyperparameters *weights for the type of mutations*. Another method of *Tree* is the method `get_transfer`, it calls the method `get_equivalent`. This method illustrates the advantages of object-oriented programming. It first calculates the equivalent T-circuit of the whole *Tree* and then its transfer function. The method does this by going to the top of the *Tree*, see the Psuedo code in Listing 4.1. If the top of the *Tree*, is a *GenotypeCircuit* (T-circuit), the method `get_equivalent` of the class *GenotypeCircuit* is called and the impedances of the equivalent circuit are returned. If the start of the *Tree* is a *GenotypeNode* (node), the method `get_equivalent` of the class *GenotypeNode* is called. This method first looks if its left branch is a node. If this is the case it goes to `get_equivalent` of that *GenotypeNode* and so on until it is a *GenotypeCircuit*. The methods `get_equivalent` of *GenotypeCircuit* returns the impedances of *GenotypeCircuit*. Then `get_equivalent` from *GenotypeNode* calculates the equivalent impedances when working its way up again by also getting the impedances of the right branch. This way the equivalent T-circuit of the whole *Tree* is calculated.

The class *GenotypeNode* is the class for a node in the *Tree*. It includes among others the type of the node (parallel, series or cascade), the depth of the node and the branches of the node. When the class is called, the type of node is chosen based on the hyperparameter *weights for the type of node*, and its branches are made. They can be either another node of a circuit. The possibility for a node is zero when the depth of the node equals the *max depth* minus one.

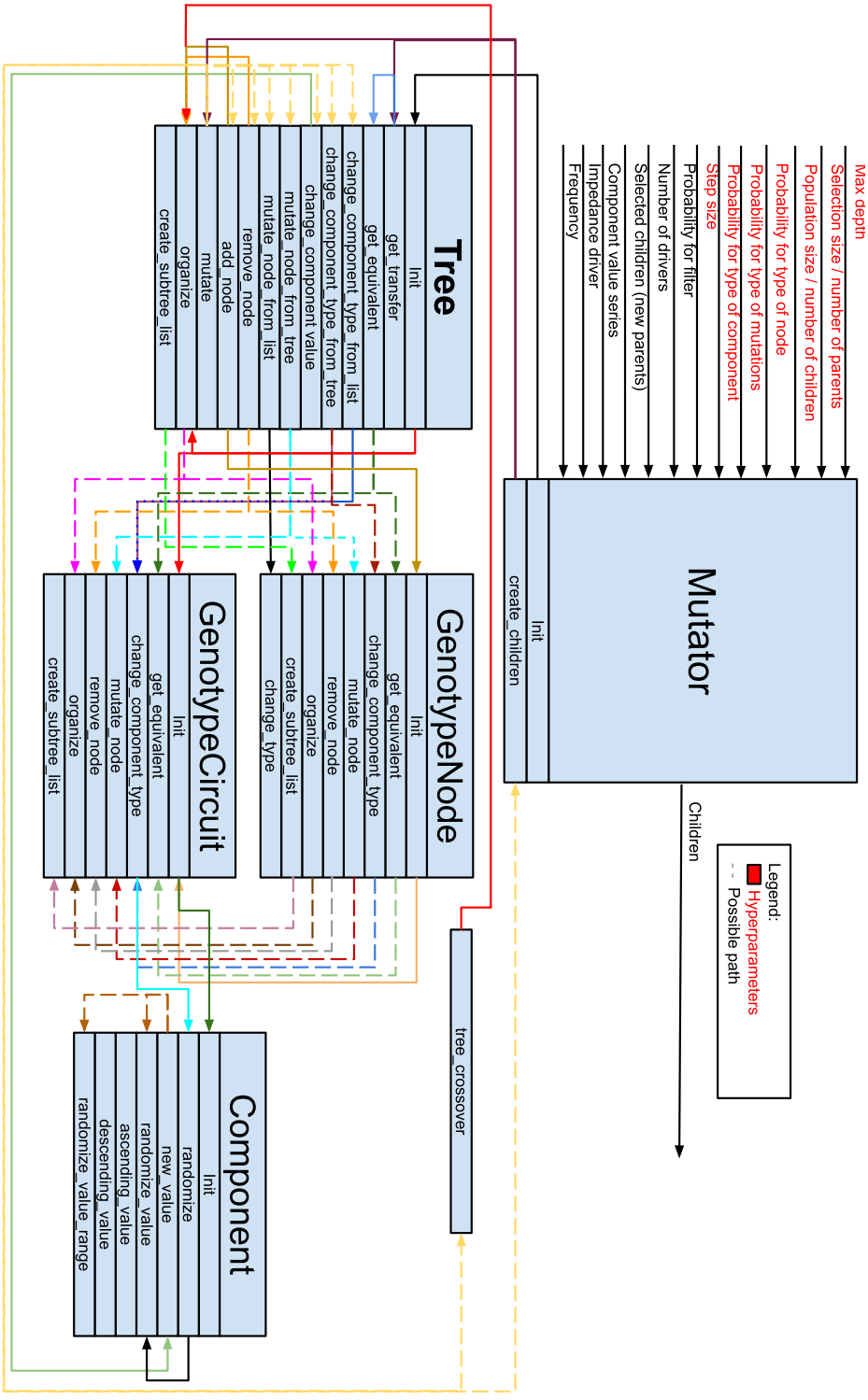


Figure 4.1: Classes and methods and their connections of the subprogram the Mutator

Listing 4.1: Psuedo code for getting equivalent circuit

```
class Tree
    def get_equivalent()
        Z[0], Z[1], Z[2] = start.get_equivalent()  #start of the tree
        transfer = Z_L / ((Z_L + Z[1]) * (1 + Z[0] / Z[2]) + Z[0])
        return z, transfer

class GenotypeNode:
    def get_equivalent()
        Za1, Za2, Za3 = branch[0].get_equivalent()
        Zb1, Zb2, Zb3 = branch[1].get_equivalent()
        if series:
            Z0, Z1, Z2 = ...
        else if cascade:
            Z0, Z1, Z2 = ...
        else if parallel:
            Z0, Z1, Z2 = ...
        return Z0, Z1, Z2

class GenotypeCircuit:
    def get_equivalent()
        Z0 = component0
        Z1 = component1
        Z2 = component2
        return Z0, Z1, Z2
```

The class *GenotypeCircuit* are the leaves of the *Tree* and is the class for the T-structured circuits. One of its attributes is a list of its three *Components*.

The class *Component* includes the type of a *Component* and its value. When the class is called, the component receives a type and a value.

4.2. Subprogram validation

4.2.1. Mutations

The validation of the functions for the mutations was done by inspecting trees before and after the mutation. An example of a textual representation is given below, which is how a tree is displayed when printed.

```
Tree:
:Node 1; Depth 1; Type: >>
    :Node 2; Depth 2; Type: >>
        :Circuit 1; Depth 3: Type LRC
        :Circuit 2; Depth 3: Type CLR
    :Node 3; Depth 2; Type: ||
        :Circuit 3; Depth 3: Type LRC
        :Circuit 4; Depth 3: Type CLR
```

After this tree was generated, the mutation `change component type` is performed on the tree. As can be seen below, a random Component changed type: the capacitor of circuit 4 was changed to a resistor. This is the same mutation as in Figure 3.10.

```
Tree:
:Node 1; Depth 1; Type: >>
    :Node 2; Depth 2; Type: >>
        :Circuit 1; Depth 3: Type LRC
        :Circuit 2; Depth 3: Type CLR
    :Node 3; Depth 2; Type: ||
```

```
:Circuit 3; Depth 3: Type LRC
:Circuit 4; Depth 3: Type RLR
```

Using this method, all mutations could be validated in a controlled environment.

4.2.2. Transfer function

It is crucial that the transfer function is correctly calculated for the filters. That is why it was thoroughly tested. To test the code for producing the transfer function, a filter would be generated together with its transfer function. The filter would then be built in LTspice and the result of LTspice would be compared with calculated transfer function. The calculation of the transfer function was tested using three methods: first, by setting the load impedance to a constant in the code. Secondly, by using the measured driver impedance and simulating in LTspice with multiple load impedances. And thirdly, by modelling the impedance of the driver with an electrical circuit.

The simplest way to verify that the transfer function that is calculated by the code is correct, is by setting the driver impedance to a constant. This means that the output transfer functions will be made with a load of constant impedance. A constant load is easily simulated in LTspice. This method of validation was used when the first results were created. An example of this approach can be seen in Figures 4.2 and 4.3. In Figure 4.2 the filter that is being tested is shown and in Figure 4.3 the calculated and simulated transfer functions can be seen.

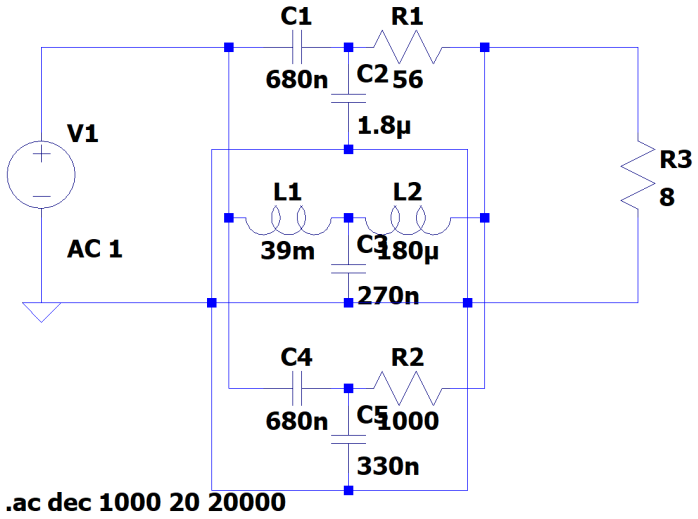
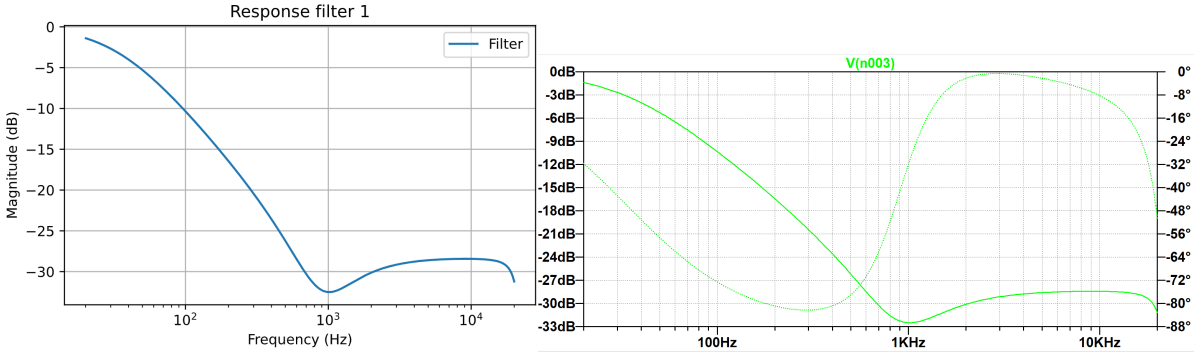


Figure 4.2: The filter under test in LTspice.



(a) Calculated with the code

(b) Simulated with LTspice

Figure 4.3: The transfer function as calculated and simulated

Once validation of the transfer function with a constant load was concluded, a different method was used for validation with a variable load. The method to test the transfer functions with the impedance

characteristics of the driver is simulating the filter with different loads. Two load impedances were chosen: one at the DC resistance of the driver and one at resonance impedance of the driver. The resulting graph would contain two traces and could be compared with the calculated transfer function. The general shape of the graphs could be compared, as well as the gain at the frequencies where the driver impedance would be exactly equal to one of the two chosen load impedances. An example of this method is given below: A filter is shown in Figure 4.4 which is simulated with two different load impedances (Z_L), 5Ω and 10Ω. The two load impedance correspond to the DC impedance and the resonance impedance of the driver. The impedance of the driver is shown in Figure 4.5. In Figure 4.6 can be seen that the general shape of the calculated transfer function corresponds with the simulation and that at the frequencies where the impedance of the driver is 5Ω or 10Ω the gain is the same.

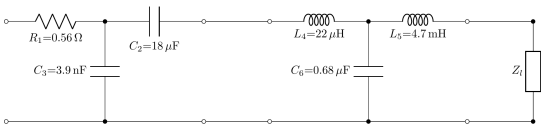


Figure 4.4: The filter for the mid-range driver under test.

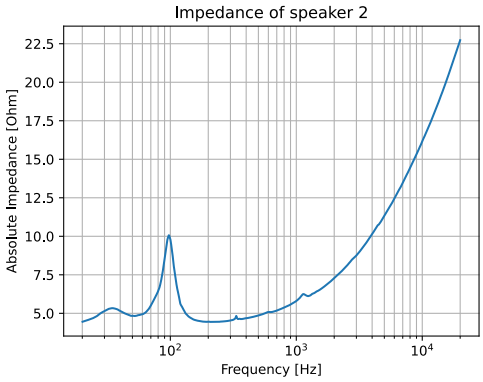


Figure 4.5: The impedance of the mid-range driver.

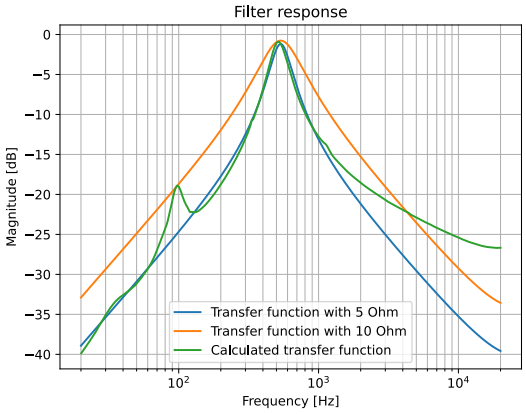


Figure 4.6: The simulated transfer function with two different load impedances and the simulated transfer function.

A second method of testing with variable load impedance, was to model the electrical characteristics of the driver. This was done using the simplification given by [4]. The electrical model of the woofer which was used for testing is shown in Figure 4.7. The impedance of the model and the actual impedance of the driver are shown together in Figure 4.8. It can be seen that the inductance of the coil (L1), makes the impedance increase too quickly and at too high frequency. This means that when looking at LTspice simulation of the filters the transfer function is less accurate at higher frequencies. However, if the code models the transfer function around the resonance peak the same as LTspice, it is assumed that the code for calculating the transfer function works as intended.

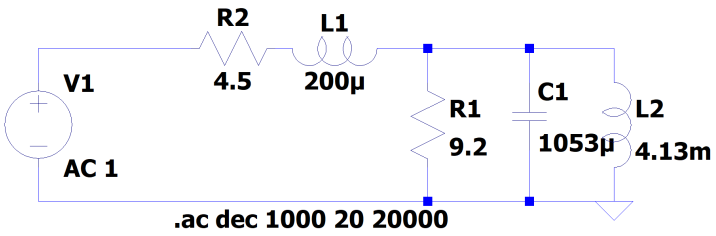


Figure 4.7: The electrical model for the driver impedance in LTspice.

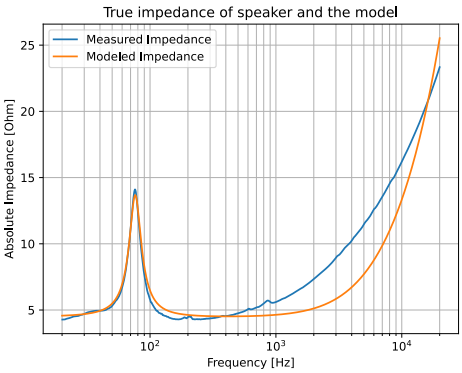


Figure 4.8: The measured impedance of the woofer and the impedance of the model.

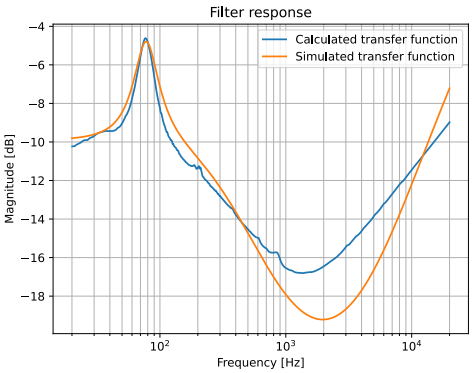


Figure 4.9: The simulated and calculated transfer functions of the filter shown in Figure 4.10

The results of the validation test of the filter shown in Figure 4.10 are shown in Figure 4.9. The model and the measured plots correspond well around the resonance peak. At frequencies where the model and the real driver have the exact same impedance, like for example at 400 Hz, the gain of the filter is exactly the same. The differences in the transfer function between the model and the code can be explained by the approximation of the electrical model of the driver.

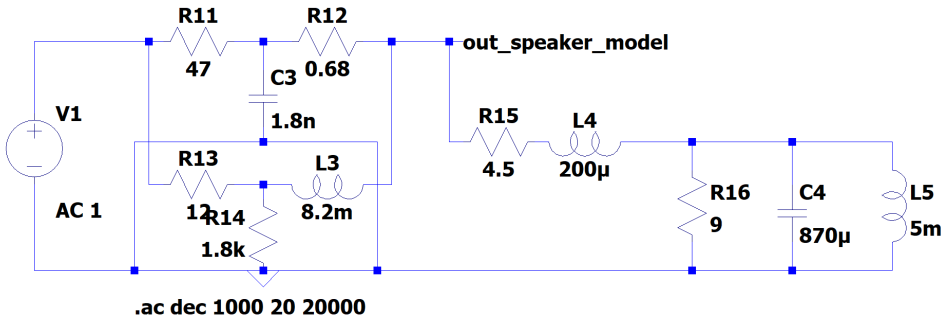


Figure 4.10: The filter under test with the electrical model of the driver as load

4.2.3. Embryo

If the program has to do fewer mutations to reach its final filter, an embryo circuit is useful. As the component values of an embryo circuit are chosen randomly, the places of the type of components are most important to determine if an embryo circuit is useful. An embryo circuit appearing in the final filter will indicate that the embryo circuits are constructed in a useful way, as the program had to do less mutation to get that specific T-circuit in the filter. In the following output code, it can be seen that the embryo circuit (RRC) fot the filter for the bass is present in the best child after 100 generations.

```
Tree:
    :Node 1; Depth 1; Type: >>
        :Circuit 1; Depth 2: ; Type LLC
        :Node 2; Depth 2; Type: ||
            :Circuit 2; Depth 3: ; Type LRC
            :Node 3; Depth 3; Type: --
                :Circuit 3; Depth 4: ; Type RRC
                :Circuit 4; Depth 4: ; Type LRL
        :Circuit 1; Depth 2: ; Type LLC
```

More trials were done to determine whether these embryo circuits are useful. In total 12 runs were done, each of one hour. After each run, the final filters for each driver of the best child were printed. In these filters, the number of embryo T-circuits present was counted. The results are shown in Table 4.1.

The probability of a specific T-circuit topology that is present in a final filter is the probability of specific topology times the number of T-circuits present in the final filters. There are three components and three places for components in a T-circuit. Therefore, there are $3^3 = 27$ different T-circuit topology. These trial runs were done with the hyperparameters *weights for a component type* set to all equal weights. Therefore, the probability for each T-circuit topologies is equal, $\frac{1}{27} \approx 3.70\%$. In the last column of Table 4.1, the percentages of the number of embryo circuits present compared with the total number of T-circuits are shown. Comparing the probability of a specific T-circuit to these numbers, only the embryo circuit for the filter for the tweeter is functional as it appears more often in the final filter than if it was chosen randomly. Therefore, the other embryo topologies are discarded and replaced with randomly chosen topologies. The embryo circuits for the filter for the tweeter are kept.

Table 4.1: Test runs for validation embryo circuits

Max Depth	Driver	Total number of embryo T-circuits in the final filter in 6 runs	Total number of T-circuits in 6 runs	Percentage of embryo T- circuits present
4	Bass	1	48	2.08%
	Mid-range	0	48	0.00%
	Tweeter	2	41	4.88%
5	Bass	1	94	1.06%
	Mid-range	3	96	3.13%
	Tweeter	4	93	4.30%

4.2.4. Hyperparameters

In this section, the values chosen for the hyperparameters *max depth* and *weights for the type of mutation* will be validated. The values for the hyperparameters number of parents and the number of children are determined by the subgroup Evaluation. The value for the *step size*, *weights for the type of node*, and *weights for the type of component* are explained in Section 3.5.5, 3.5.3, and 3.5.2, respectively.

4.2.4.1. Max depth

The *max depth* was tested by running the program for one hour, on two different laptops (one with clock speeds of the CPU of 1.60 GHz and the other of 2.60 GHz), in total 12 times. Six of these runs were performed with a *max depth* of four and six with a *max depth* of five. The results are shown in Table 4.2. Two important conclusions can be drawn from these results. The first is that a higher *max depth* does indeed result in less number of generations that can be executed during a run of one hour. The second is that even though fewer generations are run, the cost of the best child is found to be significantly better with a *max depth* of 5 than of 4.

4.2.4.2. Weights for the type of mutation

Twelve test runs were executed to validate that the *weights for the type of mutation* are more optimum than having no weights. Six runs were executed with the weights of Table 3.4 and six with equal weights.

Table 4.2: Test runs for validation max depth

Max depth	Run	Number of generations	Clock speed of CPU (GHz)	Cost of best child
4	1	2973	1.60	5.37
	2	2593	1.60	4.66
	3	2493	1.60	2.34
	4	3477	2.60	2.28
	5	3544	2.60	2.00
	6	3269	2.60	4.61
5	7	1874	1.60	1.80
	8	1895	1.60	1.52
	9	1823	1.60	0.83
	10	2363	2.60	1.16
	11	2512	2.60	1.31
	12	2528	2.60	1.32

The resulting costs are shown in Table 4.3. After the last generation, the program optimizes the values of the components (done by the Controller [23]). The value before this optimization is chosen in order to compare only the effects of the mutations. The average cost of the best child before optimization of component values for the runs with equal weights is approximately 2.98. For the runs with non-equal weight, this is approximately 2.60. This favors the weights as in Table 3.3. However, the equal weights are chosen for the type of mutation for two reasons. The first is that after running more generations, run 4-6 and 10-12, the costs of the best child are approximately equal. And because the program will do at least this number of generations, this should be taken into account. The second reason is that, if not significantly beneficial for the program, leading the AI should be avoided. Therefore the final *weights for the type of mutation* are all set to one. As child crossover was later implemented the weight was not taken into account doing these test runs.

Table 4.3: Test runs for *weights for type of mutation*

Weights	Run	Number of generations	Clock speed of CPU (GHz)	Cost of best child before optimization component values
Equal weights	1	1840	1.60	2.84
	2	2006	1.60	3.44
	3	1848	1.60	4.01
	4	2510	2.60	2.68
	5	2383	2.60	2.56
	6	2334	2.60	2.33
Weights as in Table 3.4	7	1946	1.60	2.27
	8	1795	1.60	2.66
	9	1795	1.60	2.80
	10	2237	2.60	2.41
	11	2124	2.60	2.66
	12	2124	2.60	2.77

4.3. Program validation

To conclude this chapter, the results of the best run thus far are shown. This is run 9 of Table 4.2. Each filter response and the acoustic response of each driver, combined with the filter, as well as the overall acoustic response of the loudspeaker system is shown in Figure 4.11. It can be seen that the final acoustic response of the load speaker system stays within the required margin of $1.5dB$, except for a few small spikes. The runtime was one hour, with 1823 generations, for an average of 2 seconds per generations. The filters circuits are shown in Appendix B.

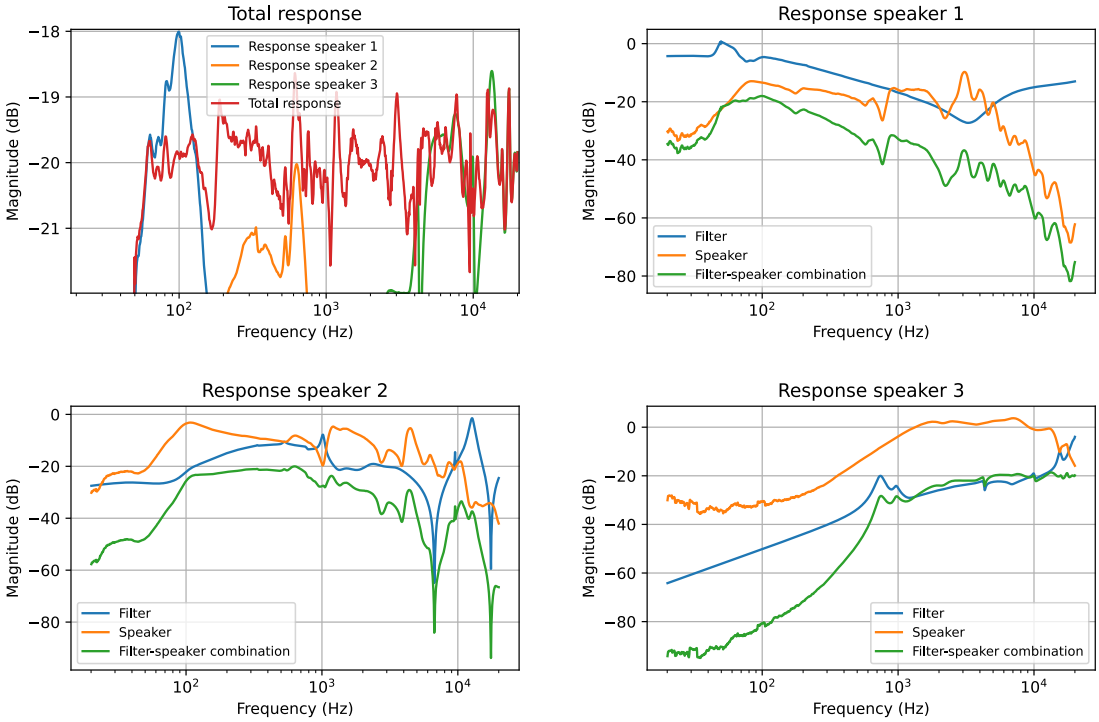


Figure 4.11: The final result

5

Discussion and Conclusion

5.1. Discussion

Having validated that the data structure, mutations, calculation of the transfer function, embryo circuits, and hyperparameters operate as intended, the program of requirements must be referred back to, to see whether they are met.

From the program of requirements, the mandatory requirements have all been met. To elaborate on a few: Requirement 11 was implemented, but removed again. Requirements 11 and 12 are not outputs, but can be read from the Mutator class.

The results of the program show that using computers to design filters for loudspeakers is not hindered by the implementation of the representation of circuits or finding their transfer function. This is promising, as it means that the step to having AI design speaker filters is not impossible because of computer technicality. The filters can be understood and modified by artificial intelligence. It shows that by making a smart algorithm, the goal of this project could be achieved.

A part that could be improved upon is removing some of the limitations imposed by using a tree data structure. For instance, allowing open and closed circuits as components, and having more types of connections (nodes) by making combinations of connections already used.

The hyperparameters is also a part that could be improved. The hyperparameters have been shown to make a difference, but their effect has not been fully investigated. By doing more and well thought-out testing, by for instance looking at their effect at runs of a different number of generations, improved hyperparameters could make the program more efficient.

5.2. Conclusion

The design of the mutation part of a genetic algorithm for designing loudspeaker filters is explained. The data structure chosen for filter circuit representation is a tree structure, which has the advantage of working well for crossover and it can be easily analyzed to find the transfer function. However, it has the disadvantage of being limited in the number of possible filter circuits and not being intuitive.

The tree structure is explained, together with how the mutations are performed on the trees, what embryo circuits are created and which hyperparameters were worked into the design and how these are set.

The code for creating trees is tested and output examples of a textual representation for a tree are given. The mutations have been tested and found to be working as intended. The calculation of a transfer function from a tree is explained and shown in pseudo-code, showing an implementation of functions acting on a tree. The calculation of transfer functions is shown to be correct using LTspice.

The design satisfies the mandatory requirements. The Mutator makes embryos, it has the correct inputs and outputs, and it can perform the needed operations. From the trade-off requirements some have been completed: An embryo circuit is designed for each driver, the weights for the type of mutation are variable, and the calculation of the transfer function is fast enough to be performed thousands of times per hour.

From this can be concluded that passive analog filter design can be implemented. Filter circuit representation, mutation and analysis for AI can be implemented. This paves the way to make design programs for passive filters in any application.

5.3. Recommendation

Efficiency

Testing and design of the hyperparameters *step size*, the *weights for the type component*, and the *weights for the type of node* are lacking and should be further investigated. This is something that could make the algorithm more efficient. An approach could be to run the program many times and look at the program many times using different hyperparameters, as done for *max depth* and the *weights for the type of mutation*.

Something else that could be explored is to make the choice of the mutation less random by changing the weights for the type of mutation to be dependent on, for example, the cost, the generation, or the previous mutation. The type of mutation could also be determined by working more closely together with the evaluation part of the project to look at what parts of a parent could be improved.

Improving the efficiency could also be done by making better embryo circuits. This could be done by looking at which T-circuits appear often in the final filters.

For `child crossover`, the replacing filter is chosen randomly, but this could be done by looking at the cost of the other filters.

The recommendations above all try to lead the AI in a certain direction, which can improve the runtime, but it could also degrade the quality of the solutions. Because the AI is given hints on which way to go, the creative solutions found by the randomness of the design are lost.

Features

Apart from efficiency improvements, other improvements could be made that add functionality.

An interesting approach that could be explored is designing filters with one input and three outputs, one for each driver. This could save components as some parts could be used for multiple drivers.

Something that could greatly improve the usefulness, is if the program would be able to work with tolerances and non-idealities of components. A sensitivity analysis per component would be very nice to implement.

The solution space (number of possible filter designs) could be improved, by for instance the addition of more types of nodes like series-parallel hybrid connections.

Another simple feature that could be added is making the max depth of a tree dependent on user input.

A

Transfer function

A.1. Equivalent circuit calculations

A.1.1. Cascading

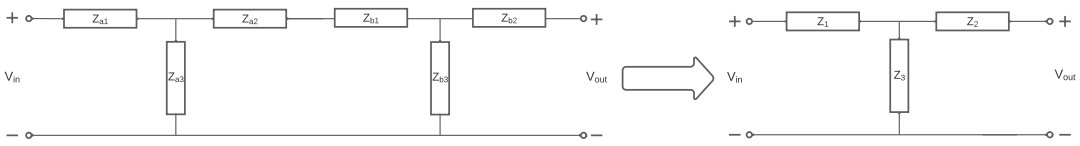


Figure A.1: Two T-circuits cascaded

$$Z_1 = Z_{a1} + \frac{Z_{a2} + Z_{b1} * Z_{a3}}{Z_{a2} + Z_{a3} + Z_{b2} + Z_{b3}} \tag{A.1}$$

$$Z_2 = Z_{b2} + \frac{Z_{a2} + Z_{b1} * Z_{b3}}{Z_{a2} + Z_{a3} + Z_{b2} + Z_{b3}} \tag{A.2}$$

$$Z_3 = \frac{Z_{a3} * Z_{b3}}{Z_{a2} + Z_{a3} + Z_{b2} + Z_{b3}} \tag{A.3}$$

A.1.2. Parallel

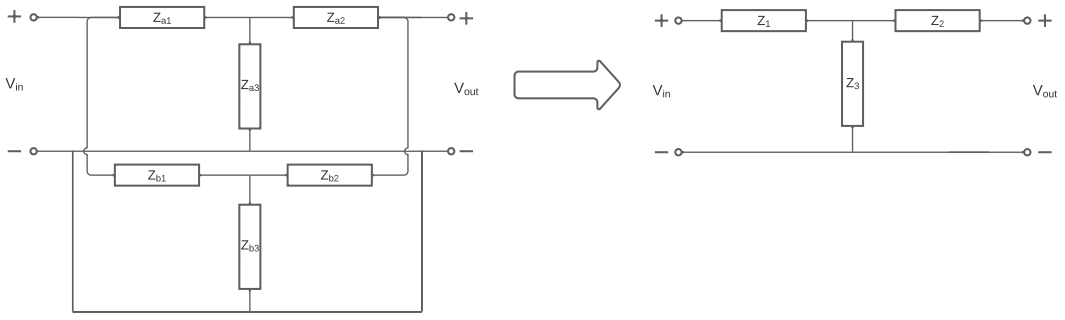


Figure A.2: Two T-circuits in parallel

$$\Pi_1 = \frac{1}{\frac{1}{Z_{a1}+Z_{a3}+\frac{Z_{a1}*Z_{a3}}{Z_{a2}}} + \frac{1}{Z_{b1}+Z_{b3}+\frac{Z_{b1}*Z_{b3}}{Z_{b2}}}} \tag{A.4}$$

$$\Pi_2 = \frac{1}{\frac{1}{Z_{a2}+Z_{a3}+\frac{Z_{a2}*Z_{a3}}{Z_{a1}}} + \frac{1}{Z_{b2}+Z_{b3}+\frac{Z_{b2}*Z_{b3}}{Z_{b1}}}} \tag{A.5}$$

$$\Pi_3 = \frac{1}{\frac{1}{Z_{a1}+Z_{a2}+\frac{Z_{a1}*Z_{a2}}{Z_{a3}}} + \frac{1}{Z_{b1}+Z_{b2}+\frac{Z_{b1}*Z_{b2}}{Z_{b3}}}} \tag{A.6}$$

$$Z_1 = \frac{\Pi_1 * \Pi_3}{\Pi_1 + \Pi_2 + \Pi_3} \tag{A.7}$$

$$Z_2 = \frac{\Pi_2 * \Pi_3}{\Pi_1 + \Pi_2 + \Pi_3} \tag{A.8}$$

$$Z_3 = \frac{\Pi_1 * \Pi_2}{\Pi_1 + \Pi_2 + \Pi_3} \tag{A.9}$$

A.1.3. Series

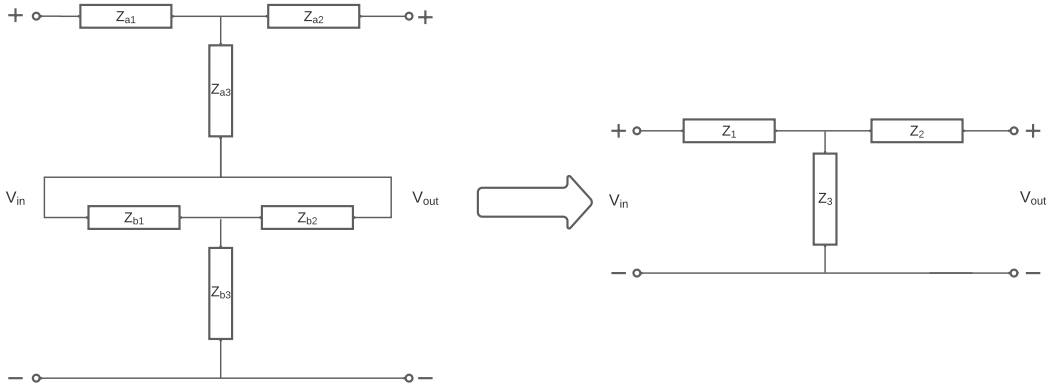


Figure A.3: Two T-circuits connected in series

$$Z_1 = Z_{a1} \tag{A.10}$$

$$Z_2 = Z_{a2} \tag{A.11}$$

$$Z_3 = Z_{a3} + \frac{1}{\frac{1}{Z_{b1}} + \frac{1}{Z_{b}2}} + Z_{b3} \tag{A.12}$$

A.2. Π-T structure transform

In this appendix, the equation for the conversion from Π to T-circuit and visa versa are given. Π-circuits are also known as Δ-circuits and T-circuits as Y-circuits.

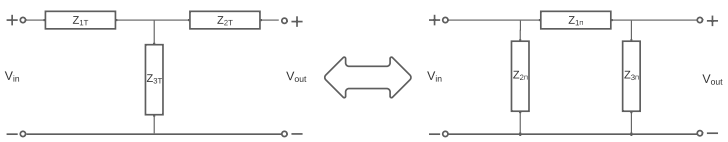


Figure A.4: Π to T-circuit conversion and visa versa

A.2.1. From T-structure to Π-structure circuit

$$Z_{1\Pi} = \frac{Z_{1T} * Z_{2T} + Z_{2T} * Z_{3T} + Z_{1T} * Z_{3T}}{Z_{3T}} \tag{A.13}$$

$$Z_{2\Pi} = \frac{Z_{1T} * Z_{2T} + Z_{2T} * Z_{3T} + Z_{1T} * Z_{3T}}{Z_{2T}} \tag{A.14}$$

$$Z_{3\Pi} = \frac{Z_{1T} * Z_{2T} + Z_{2T} * Z_{3T} + Z_{1T} * Z_{3T}}{Z_{1T}} \tag{A.15}$$

A.2.2. From Π-structure to T-structure circuit

$$Z_{1T} = \frac{Z_{1\Pi} * Z_{2\Pi}}{Z_{1\Pi} + Z_{2\Pi} + Z_{3\Pi}} \tag{A.16}$$

$$Z_{1T} = \frac{Z_{1\Pi} * Z_{3\Pi}}{Z_{1\Pi} + Z_{2\Pi} + Z_{3\Pi}} \tag{A.17}$$

$$Z_{1T} = \frac{Z_{2\Pi} * Z_{3\Pi}}{Z_{1\Pi} + Z_{2\Pi} + Z_{3\Pi}} \tag{A.18}$$

B

Final Results

Below are the three filters corresponding to the results described in 4.3. These are from before components could be removed in the final optimization from [23].

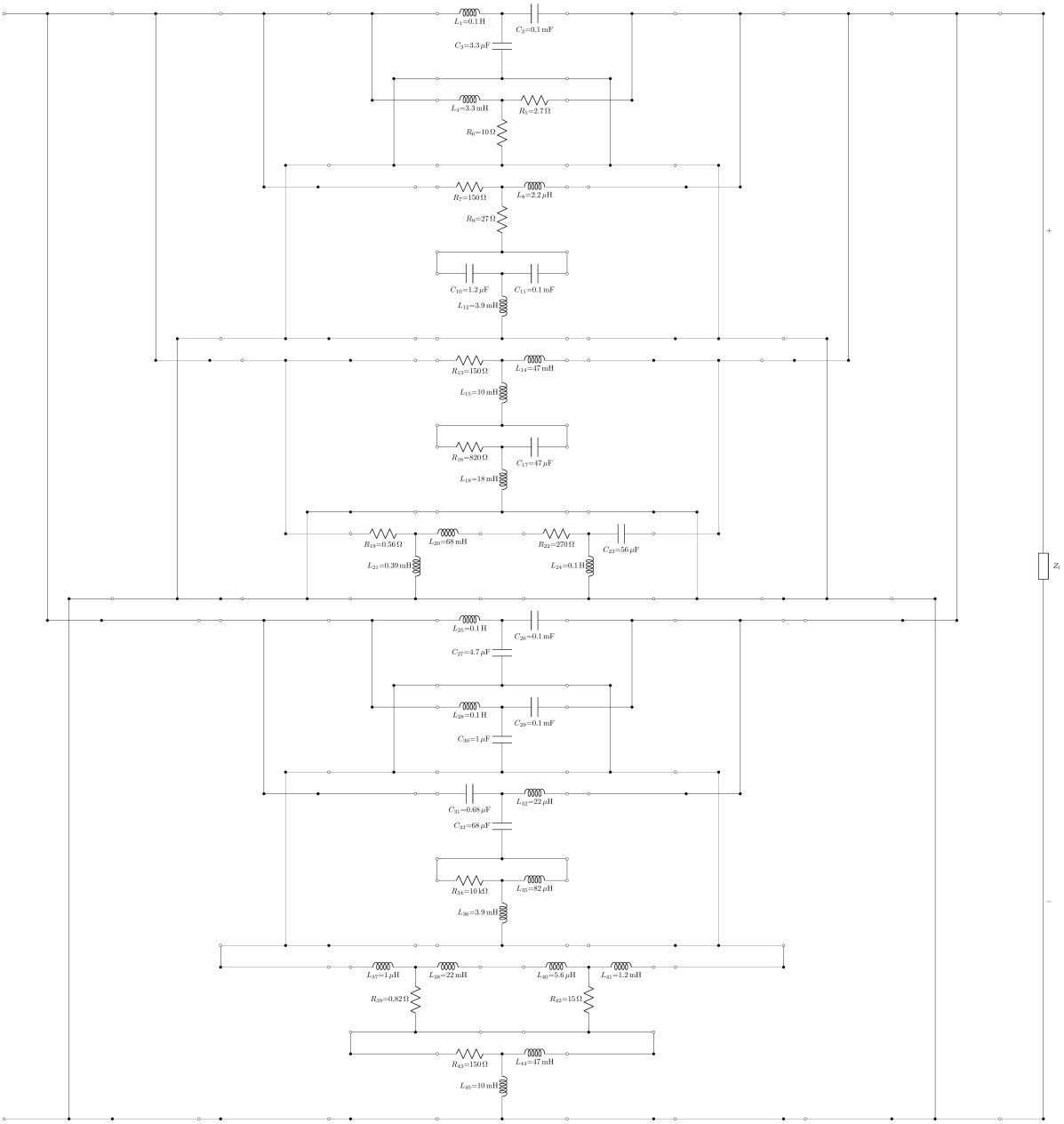


Figure B.1: The filter for the bass driver

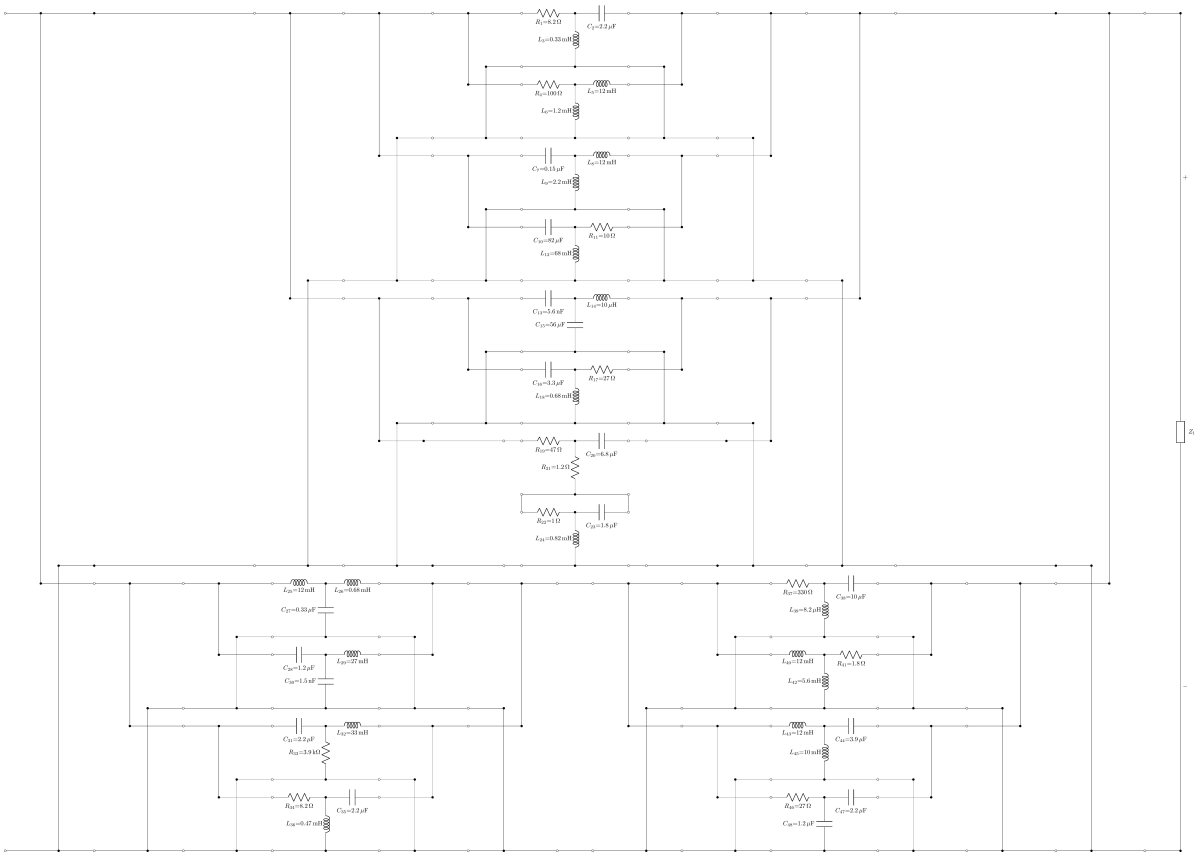


Figure B.2: The filter for the mid-range driver

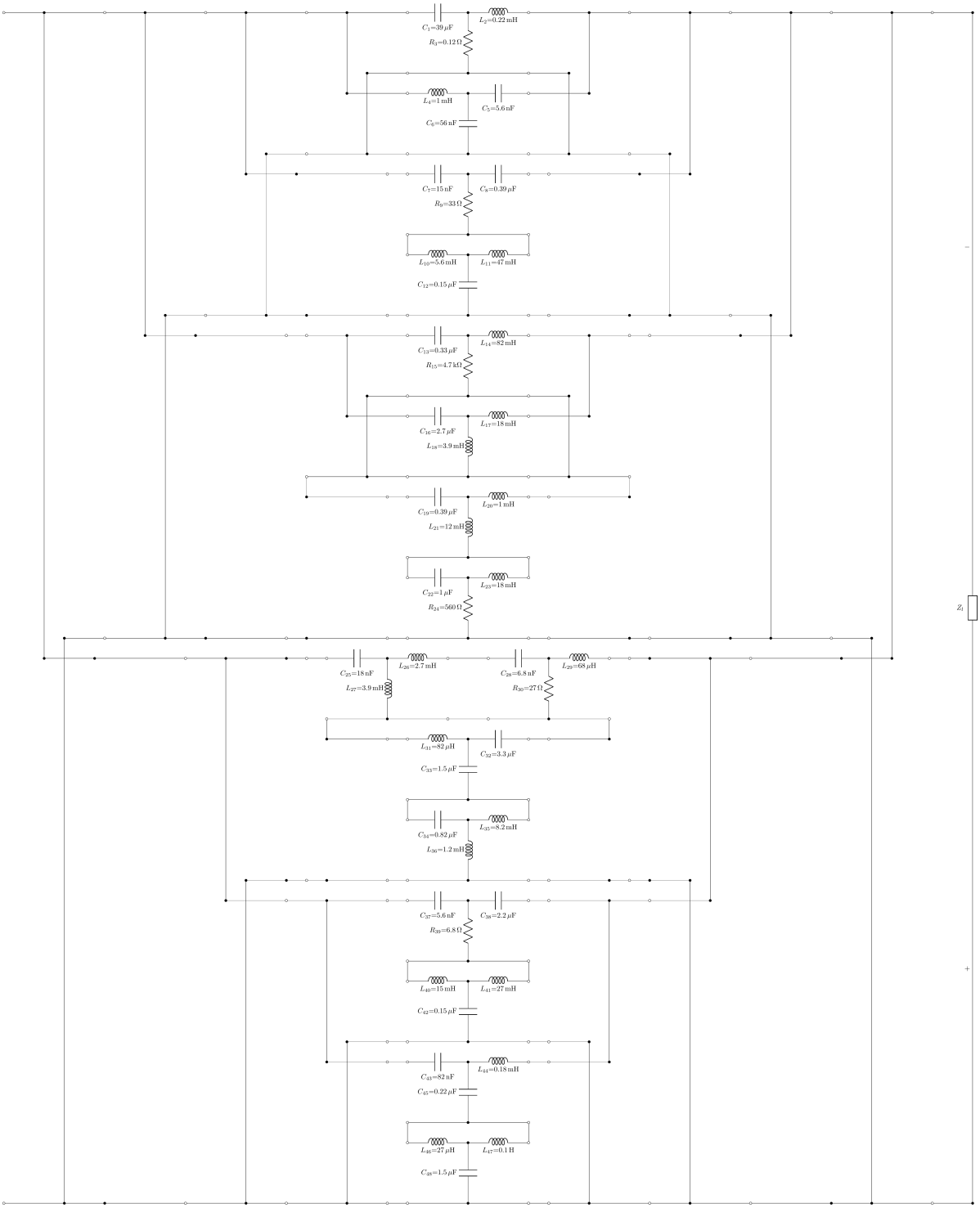


Figure B.3: The filter for the tweeter

Bibliography

[1] [Online]. Available: https://hotpot.ai/art-generator.

[2] 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.

[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.

[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, Second 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. 1 350 106, 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. 109 740, 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. 114 267, 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] F. Castejón and E. J. Carmona, “Automatic design of analog electronic circuits using grammatical evolution,” *Applied Soft Computing*, vol. 62, pp. 1003–1018, 2018.

[21] 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.

[22] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697. DOI: 10.5555/1593511.

[23] J. Nijenhuis and T. Zunderman, “Loudspeaker filter optimization with AI, Communication, evaluation, and user interaction for an AI loudspeaker filter design program,” 2023.

[24] K. Bavelaar and J. Verweij, “Loudspeaker filter optimization with AI, Genetic algorithm selection methods,” 2023.

[25] 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.

[26] O. Verducci, P. C. Crepaldi, L. B. Zoccal, and T. C. Pimenta, “Synthesis of passive filter using object oriented genetic algorithm,” in *2014 26th International Conference on Microelectronics (ICM)*, 2014, pp. 72–75. DOI: 10.1109/ICM.2014.7071809.

[27] J. He, M. Liu, and Y. Chen, “A novel real-coded scheme for evolutionary analog circuit synthesis,” in *2009 International Workshop on Intelligent Systems and Applications*, 2009, pp. 1–4. DOI: 10.1109/IWISA.2009.5072665.

[28] S.-J. Chang, H.-S. Hou, and Y.-K. Su, “Automated passive filter synthesis using a novel tree representation and genetic programming,” *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 1, pp. 93–100, 2006. DOI: 10.1109/TEVC.2005.861415.

[29] C. Spinningpark, *Cc by-sa 3.0*, Accessed: 05-06-2023. [Online]. Available: https://en.wikipedia.org/w/index.php?curid=28728942.

[30] C. Spinningpark, *Cc by-sa 3.0*, Accessed: 05-06-2023. [Online]. Available: https://en.wikipedia.org/w/index.php?curid=28755585.

[31] C. Spinningpark, *Cc by-sa 3.0*, Accessed: 05-06-2023. [Online]. Available: https://en.wikipedia.org/w/index.php?curid=11254101.

[32] M. Hayes, “Lcapy: Symbolic linear circuit analysis with python.,” 2022. DOI: https://doi.org/10.7717/peerj-cs.875.