Loudspeaker Filter Design With AI

Genetic Algorithm Selection Methods

by

Koen Bavelaar
Jeroen Verweij

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

Student numbers: 5345154, 5341752

Project duration: April 24, 2023 – June 30, 2023

TU Delft

Contents

1

Abstract

This thesis details the design of a selection operator used in a Genetic Algorithm. The Genetic Algorithm is used for loudspeaker filter design of three way loudspeakers for which tournament selection was chosen as selection operator. A methodology is proposed and used to tune the parameters of tournament selection, which is based on diversity and fitness of the population. Besides basic tournament selection, two new adaptive selection operators based on tournament selection are proposed to improve its functionality. The first adaptive selection operator uses noise proportional to the fitness variance of the population to improve the efficiency of the genetic algorithm. The second adaptive selection operator uses a convergence stage to speed up the convergence towards the optimal filter. After the presented tuning process in this thesis, the latter adaptive selection operator was found to perform better. The optimal selection operator and parameters found in this thesis will not translate to every application, because they heavily depend on the design and the application of the genetic algorithm. However, the presented comparison of selection operators, the provided performance metrics and design methodology can still be used to guide the choice and the tuning process of a selection operator used in any genetic algorithm.

2

Preface

The goal of this thesis is to show a strategic way to determine a selection operator for a genetic algorithm. Even though the selection operator is of vital importance for the performance of the genetic algorithm, general steps to select an operator and its parameters are not available. Usually trial and error methods are used to determine a suitable selection operator and its parameters. We hope that the design process presented in this thesis provides a more structured approach to determine a suitable selection operator and its parameters. We would like to thank Willem van Overbeeke, Zoyla Barendse, Jonathan Nijenhuis and Timo Zunderman for the pleasant collaboration during the project. In addition we would like to thank Jonathan Nijenhuis and Timo Zunderman for their extensive help in using the Python programming language. We would like to thank Justin Dauwels for his advise during the green light assessment and of course Gerard Janssen for his extensive guidance during the project.

Koen Bavelaar & Jeroen Verweij
Delft, June 2023

3

Introduction

In a world where Artificial Intelligence (AI) is becoming more and more relevant and new applications are invented every day, AI has been used to come up with better solutions for problems where only minor improvements were made in the last few decades. In the field of designing analog filters, AI is able to come up with multiple solutions that fulfill 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.

3.1. 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 [11], 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.

3.2. 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 [1, 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 [8, 38]. 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 [31]. 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.

[26] 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 [19, 35], simulated annealing and genetic algorithm (e.g. [9, 16, 37]). 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 [9, 22] and designing a topology and optimizing the values [23, 29]. When optimizing values for components, both optimizing for ideal values and discrete values is possible [22]. 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 [7, 13, 17, 18, 30]. In [13], the simulation takes around 90 percent of the total computation time, the exact percentage depending on the complexity of the circuit. [18] 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.

3.3. 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 [34]. 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 [7, 9, 13, 16, 17, 18, 37]. The theory of genetic algorithms (GA's) was originally developed and published by John Holland in 1975 [20]. A genetic algorithm is based on Darwin's theory of natural selection, which is the process where a population of individuals changes over time, due to promotion of individuals that have beneficial traits with respect to their environment. A genetic algorithm is an adaptive heuristic search algorithm, which comprises of some general steps as shown in the flowchart 3.1.

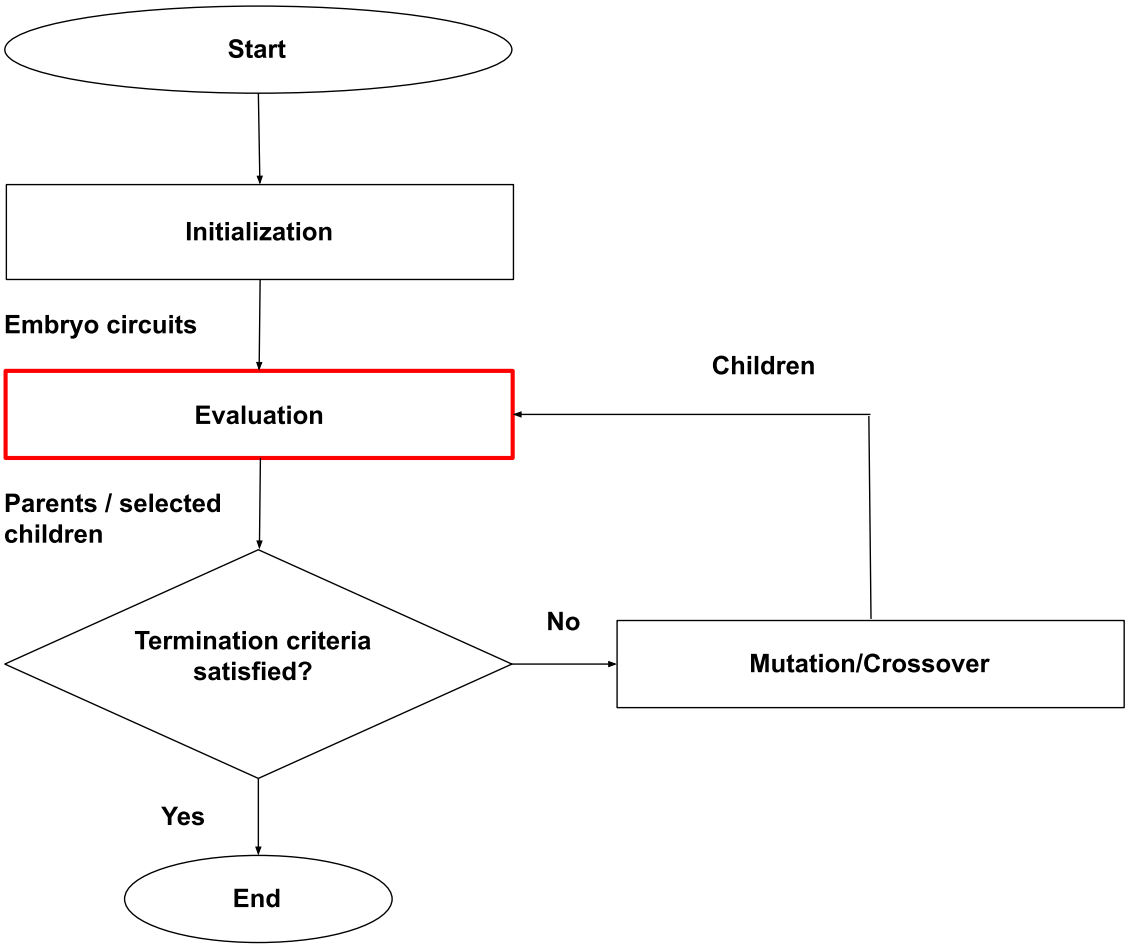


Figure 3.1: Flowchart genetic algorithm

The GA starts with an initialization step, where an initial population is formed. Subsequently the individuals in the population are given a score, by means of a cost function. This score is then used to evaluate which individuals in the population qualify as parents or children during the next generation. The selected parents are crossovered and mutated. The selected children and mutated and crossovered parents are then evaluated again. This process continues until the predefined termination criteria are satisfied.

In the case of a three-way speaker, the initial population consists of three circuits per child, one circuit for each driver, called embryo circuits. 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 operator.
- 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 3.2, a block diagram can be found where the signals with data between each subgroup have been drawn.

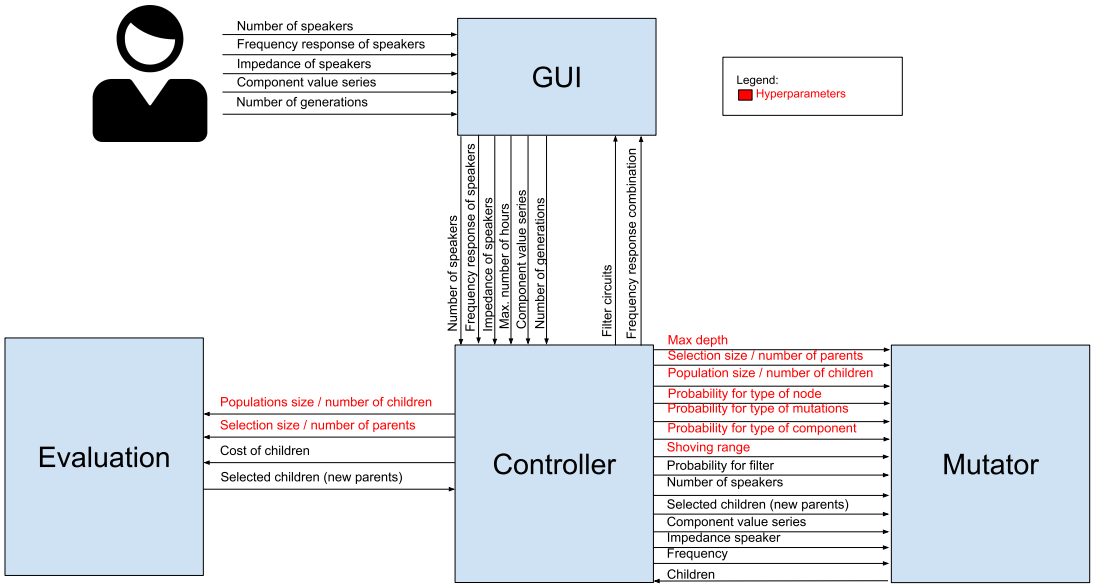


Figure 3.2: A diagram with the subdivision and the signals between the groups

3.4. Problem definition & goal of the project

In this thesis the evaluation step of the genetic algorithm will be considered. As already stated above, the evaluation step is responsible for the selection of individuals from a population. In the context of the application of the GA, the population consist of predefined number of filter sets, where each filter in the set is used for one speaker driver. Each set of filters is given a score by the controller based on their performance, which is evaluated by means of a cost function. Based on the assigned scores sets of filters are selected. This selection is realized by means of a selection operator. The goal of this thesis is to find the optimal selection operator in order to efficiently converge the population towards the optimal filter set, which satisfies the predefined amplitude response criteria.

3.5. Structure of thesis

This thesis is structured as follows. In chapter 4 the imposed requirements to the project are presented. Chapter 5 provides a state of the art analysis regarding the selection operator. In chapter 6 several selection operators are compared in order to determine which is the most suitable for the imposed application. Chapter 7 elaborates on the concept of diversity and proposes methods to measure this attribute. Chapter 8 describes the methodology which is used to tune the parameters of the selection operator. Chapter 9 is concerned with the tuning process of the chosen selection operator. In chapter 10 two novel selection procedures are proposed and tested. Chapter 11 gives the conclusion of the project and future work.

4

Program of requirements

4.1. Requirements for the complete program

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

4.1.1. Mandatory Requirements

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

Program requirements

1. The program must take the frequency response of the individual drivers of a three-way loudspeaker system as an input.
2. The program must take the impedance of the individual drivers of a three-way loudspeaker system as an input.
3. The program must identify constraints for the filters based on the frequency responses and impedances as specified in Section 4.1.1.
4. The program must be able to design a passive analog filter for each driver with a minimal performance as specified in Section 4.1.1.
5. The runtime of the program must not exceed 12 hours on an HP ZBook Studio G5 with an Intel Core i7-9750H CPU at 2.60 GHz.

Filter constraint requirements

6. The program must determine an operating range of each of the drivers, using the constraints from Section 4.1.1, 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 4.1.1.

4.1.2. Trade-off requirements

The requirements in this section of the PoR would improve the usability of the final program. These requirements are not mandatory, but nice to have. The trade-off requirements, shortly ToR, are not

SMART formulated, since it is not necessary to fulfill these requirements in order to have a working program. The ToR consist of three sets of requirements: for the program, the user and the filter.

Program requirements

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

User requirements

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

Filter requirements

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

4.2. Requirements for the evaluation group

In this section the requirements for the evaluation group are stated. During the course of the project the approach to the project has changed. Some objectives were impossible or did not contribute to the achievement of predefined goals of the program, so not all mandatory requirements are still relevant. The changes are covered later in section 4.2.3.

4.2.1. Mandatory requirements

1. The input of the evaluation process must be a list of costs of the individuals in the population.

2. The output of the evaluation process must be a list of selected parents.

3. The chosen selection operator must minimize the cost to meet the predefined amplitude response within 12 hours.

4.2.2. Trade-off requirements

1. The run-time of the program should be minimized.

2. The fitness of the created circuit should be maximized.

3. The chance of success of a run should be maximized.

4. The program should send output flags to the Mutator that describe what change could be beneficial for the circuit.

4.2.3. Changes

A few requirements have been removed:

- The input of the evaluation process must contain a defined number of symbolic transfer functions.
- The input of the evaluation process must contain component value lists for the symbolic transfer functions.
- The output of the evaluation must consist of a signal that notifies the controller when a child satisfies the constraints specified by the controller.
- The evaluation process must rank the circuits based on the cost function and potential.
- The input of the evaluation process must contain the SPL of the individual speakers.
- The input of the evaluation process must contain the operating range of individual speakers.

The symbolic transfer function including the component value list can not be received from the Mutator group. This is due to the computationally expensive nature of symbolic transfer functions, calculating the symbolic transfer function would slow down the program to such an extend it would become unusable in practice. Also, the Controller group now stops the program when the constraints are met because the Controller makes the cost function and has access to the constraints. The potential, and the operating range and SPL that were received to calculate potential, is no needed because potential was needed to improve the run-time of the program. Run-time is no longer a problem as the program is able to create working circuits in much less than 12 hours.

5

State of the art analysis

Choosing the proper selection operator for a GA is an important task. The effectiveness and the efficiency of a genetic algorithm is largely determined by the selection operator [3, 4]. Without a selection operator the genetic algorithm will be solely a random process. Due to the selection operator, individuals with a higher fitness have a higher probability to be selected as parent or child. This enables the algorithm to promote fit individuals and exploit their good characteristics in order to converge to the optimal circuit, which satisfies the predefined criteria.

In the literature several selection operators are used. The most important selection operators that are used nowadays are enumerated below, their properties are described in chapter 6.

1. Roulette Wheel Selection (RWS)
2. Linear rank (LRS)
3. Exponential rank (ERS)
4. Tournament Selection (TOS)
5. Truncation Selection (TRS)

Despite decades of research there are no general guidelines for selecting an effective selection operator for a specific application [21]. The literature, on the other hand, does provide guidance on which factors are important with regard to the choice of a selection operator.

According to the literature, two important factors need to be taken into account when choosing a selection operator, namely exploration and exploitation. Exploration of the search space is necessary in order to find the optimal solution and prevents convergence towards a local minimum. Search space, including local and global minima, is explained in the appendix 13.1. Exploitation focuses on exploring the environment near the most promising solutions in order to converge to a minimum. The trade off between exploration and exploitation is critical for the performance of the selection operator [6].
Two factors that determine the balance between exploration and exploitation are diversity of the population and selection pressure [28]. A more diverse population enables simultaneous exploration of the search space at different locations. With a higher diversity in fitness the GA has a more exploring character. Diversity can be measured by looking at the cost or structure of individuals in the population. In the presented application of the GA, the individuals are represented as rooted unordered binary trees and a diversity function will be created for this structure. In chapter 7 a diversity function is represented which is later used as figure of merit in order to compare different selection operators.

The term selection pressure is widely used to characterize the strength of emphasis on the selection of the fittest individuals [3]. High Selection pressure however, reduces the diversity in the population. This reduction in diversity can cause premature convergence and prevents the GA from finding the optimal solution. In order to prevent premature convergence, many methods have been proposed [27].

For example, two methods are crowding and incest prevention. The crowding method prevents premature convergence by eliminating the most similar individual whenever a new one enters the population. Incest prevention impedes premature convergence by only allowing crossover when the traits of two selected parents differ enough.

6

Determining the selection operator

In this chapter five often used selection operators are reviewed. One of these selection operators will be implemented in the GA. Due to the fact that there is not one selection technique that works best for all applications, a selection technique has to be chosen based on how well its properties fit the application, which in this case is finding the optimal filter set for a three-way speaker. All five techniques shown in table 5 will be compared according to the literature and the mathematical models described in [6].

6.1. Performance metrics

As stated in chapter 5, exploitation and exploration of the search space are important when considering a selection operator. In this section performance metrics are presented for both concepts.

6.1.1. Exploitation

[6] presents a metric that evaluates how much a selection operator exploits the search space. This metric is called selection intensity and is calculated by the following formula 6.1.

$$Selection\,pressure = \frac{E[M*] - M}{\sigma} \tag{6.1}$$

M represents the average fitness of the population before the selection procedure. $M*$ represents the average fitness after the selection procedure. σ is the standard deviation of the population before the selection procedure.

6.1.2. Exploration

[6] presents metrics that evaluate how much a selection operator explores the search space. The first metric is called ”loss of diversity”. This is the proportion of individuals that are not selected in the selection phase. This is not a good metric to determine the exploratory character of a selection operator, because it does not take into account which individuals are selected. If a selection operator selects individuals with a large variety in fitness, it may still preserve diversity even though the ’loss of diversity’ is high.

Another presented metric is called selection variance, which is calculated using the following formula 6.2.

$$Selection\,variance = \frac{E[(\sigma*)^2]}{(\sigma)^2} \tag{6.2}$$

Where $E[(\sigma*)^2]$ represents the expected variance of the population after selection. σ represents the variance of the population before the selection. This metric gives an indication of how well a selection operator preserves diversity in cost.

6.1.3. Utilization

The performance metrics elaborated above will be used to guide the decision of a selection operator. The presented source [6] evaluates and compares the metrics of different selection operators, assum-

ing that the population has a Gaussian fitness distribution. In order to check if this is indeed the case, a fitness distribution during a run of the GA was computed. The result is shown in figure 6.1. As shown in the figure, the fitness values are not normally distributed. Due to the thorough comparison of the different selection operators provided in this source based on this distribution, the results will still be used to choose a selection operator.

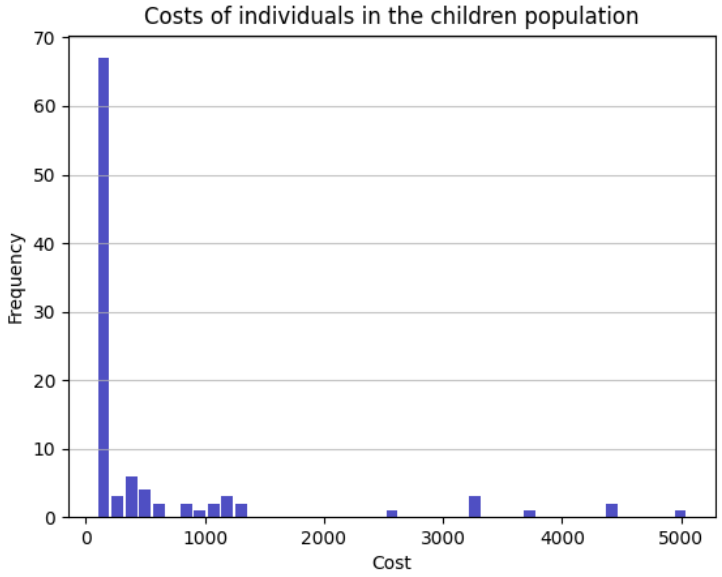


Figure 6.1: Fitness distribution

6.2. Roulette Wheel Selection, linear rank selection, exponential rank selection

The first selection operator that is considered is roulette wheel selection (or proportionate selection). It is the original selection operator proposed by John Holland [20]. The probability that an individual is selected is linearly proportional to the fitness of the individual [6]. The probability that an individual is selected can be calculated according to formula 6.3 as derived in [32]:

$$P(a_i) = \frac{\phi(a_i)}{\sum_{i=1}^{n} \phi(a_j)}, j = 1, 2, \ldots, N \tag{6.3}$$

As shown by the formula, the probability of selection of an individual is determined by its fitness with respect to the total fitness of the population. The selection probabilities however strongly depend on the the scaling of the fitness function. This is due to the fact that proportionate selection is not translation invariant [6]. If for example a constant is added to all fitness values, the probability distribution changes. In order for the selection operator to work properly a scaling method is needed [15]. Proportionate selection technique also suffers from the risk of premature convergence to a local minimum due to the possible presence of a super individuals that dominate the selected population [21]. Super individuals are individuals with a much higher fitness than the rest of the population. To prevent this, a new selection technique was proposed called linear rank selection [5]. It is a selection operator which considers the rank of the individuals instead of their fitness. The rank of an individual is the rank of fitness in the population. The individual with the best fitness in a population is rank 1, the second best has rank 2 and so forth. Also a newer adaptation called exponential rank selection exists. Exponential ranking differs from linear ranking due to the fact that the probabilities of ranked individuals are exponentially weighted [6]. Exponential rank selection has a higher selection variance than linear rank selection for almost all selection pressures which means it considers a wider range of possible circuits for the same improvement in fitness [6].

6.3. Truncation Selection

Another rank based selection operator is truncation selection. This technique ranks the individuals on fitness and selects a predefined number of highest rank individuals. It can achieve a really high selection pressure, but not a high selection variance [6]. A high selection variance is important in order to find the optimal filter set because the search space has local minima. Therefore Truncation selection will not be used in this project.

6.4. Tournament Selection

Another widely used selection algorithm is tournament selection. This selection method consist of two stages: a sampling stage and a selection stage. During the sampling stage K individuals are randomly sampled from a population of size N with replacement. These samples form a tournament group. K is called the tournament size. During the selection stage the fittest individual in the tournament group is selected for reproduction. The amount of tournaments M, determines the amount of individuals that are selected. M is therefore referred to as the selection size. In the figure below a realization of tournament selection is shown. The size of the tournament group is 2 and the amount of groups is 3.

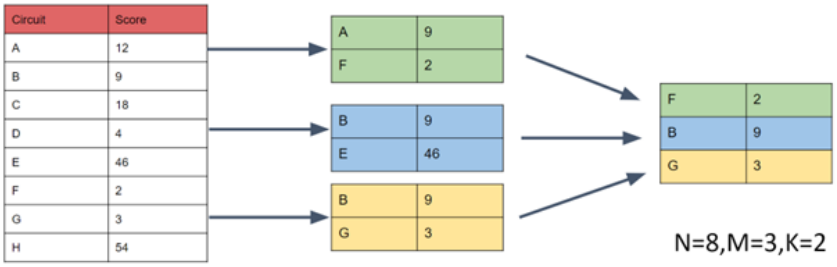


Figure 6.2: Tournament selection method

Tournament selection is often used due to the following properties [12, 36]:

1. The selection pressure can be easily adjusted by changing the size of the tournament group.
2. It is a relatively easy algorithm to program and can be performed in parallel, because the individuals are drawn with replacement from the population.
3. Sorting of the population is not needed and the algorithm has a low time complexity of O($k \cdot M$)

Genetic programming is considered as computationally intensive, so especially the last two properties made tournament selection very popular. Tournament selection can achieve a comparable performance to exponential rank selection, but can achieve a higher selection pressure for a lower selection variance [6]. There are is however a problem regarding the diversity. Due to the random selection with replacement it is possible to have individuals who never compete in a tournament [36].

6.5. Conclusion

Five different selection operators are compared. When considering the application of the genetic algorithm two factors are important. First of all it is necessary to maintain a diverse population in order to explore the search space and to prevent premature convergence. In addition it is important to provide enough selection pressure in order to converge to the optimal solution. Roulette wheel selection suffers from premature convergence problems when super individuals dominate the selection procedure. This selection operator will therefore not be implemented. Exponential and linear rank selection on the other hand, two variants of roulette wheel selection, partly solve the premature convergence problem. Exponential selection performs better than linear selection in terms of selection variance, so linear selection will not be implemented. As stated above, tournament selection is the faster than linear, exponential en truncation selection. In addition, the selection pressure can be easily tuned by changing the size of the tournament group. Based on these measures tournament selection is chosen as selection operator.

7

Diversity

As stated in the state of the art analysis, diversity of the population is an important measure that needs to be considered when designing a selection operator. It has a large impact on the effectiveness of the GA because it indicates how much the GA is exploring different minima before converging to a local minimum. Ideally the diversity shows how many different solutions are in the population. For example: A low-pass filter can be in one individual in the population while a high-pass filter is in the other. Here the GA experiments with both types of filters. If the population only has low-pass filters with slightly different component values it not exploring new options and may settle for a bad circuit while other better circuit exist. Both diversity in cost and structure have been used to determine and also influence selection pressure. Because diversity in cost is not accurate enough, as will be explained later, diversity in structure will be used. It shows how many different components and combinations of components are in a population.

7.1. General application

Diversity in cost is used to mathematically analyse selection operators [6], this was used in deciding which selection operator to use in chapter 6. Structural diversity can be used to vary selection pressure during a run of the GA. [24] proposed the method of giving individuals with rarer components a higher fitness. This will not be used in this report because it turns out very specific components turn out to be used by the GA. For example, the final circuits produced by the GA contain almost only one type of node while 3 different nodes can be used, trying to keep the ineffective nodes in the population will probably only slow down the process of finding an optimal circuit. Nodes will be explained later in this chapter. [25] Has used both diversity in cost and structure and used this to create a dynamic selection method by varying selection pressure based on the diversity of a population, the conclusion was that this works better than diversity in cost alone. In this thesis diversity will be used to tune parameters of the tournament selection method where the tuning is normally done with trial and error.

7.2. Diversity in structure

A function to calculate diversity is created that is applicable to rooted, unordered, binary tree circuit representations. This means the connections under a node can be swapped and still have the same transfer function. The tree can be seen in figure 7.1. In the genetic algorithm for this project the tree is formed with nodes of three types: cascade, parallel and series. For these tree types of nodes it is not of importance for the functionality of the circuit which circuit or node is connected left and which on the right under a node so the connections can be swapped and still give the same transfer function.

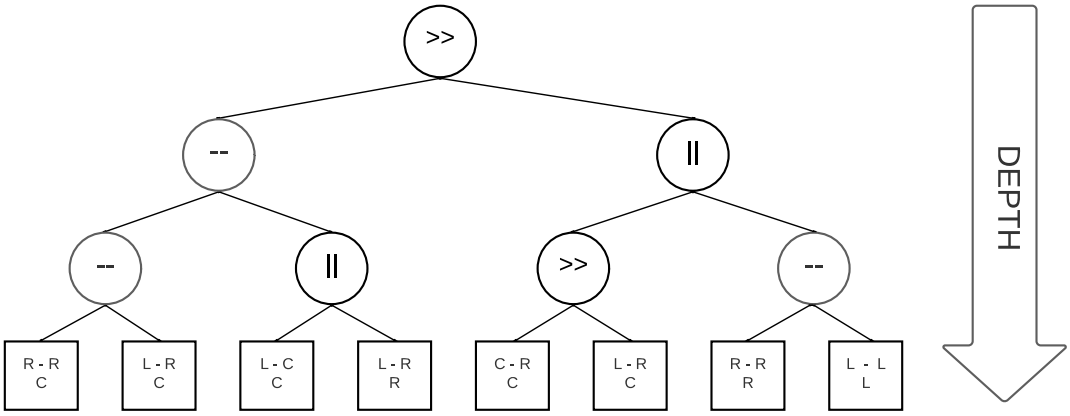


Figure 7.1: Tree

In this section the exact procedure of calculating the diversity of a circuit will be presented. It works by comparing 2 circuits (trees) and calculating how much they match. Actually not diversity but resemblance is calculated with this method. This is why the term resemblance will also be used from now on. Points are given for matching parts of the tree.

7.2.1. Individual Circuits

Matching individual circuits get 1 point. Every circuit of tree one is compared with every circuit in tree two. Points are only given if the circuit is not a duplicate. This means two circuits of type RLL in tree 1 while there is only one circuit of type RLL in tree 2 gets points only once. But if they both have two circuits of type RLL points are given twice, same for three circuits and so forth.

7.2.2. Individual Nodes

Matching individual nodes get 0 points. There are only three different types of nodes while there are 27 different types of circuits. This means that it is much more likely nodes between trees match, even though the population is diverse and exploring. Furthermore the GA shows almost exclusively parallel nodes are functional. Giving points for individual nodes would make the function inaccurate, as pure chance could give a population that is properly exploring a low diversity value.

7.2.3. Sub-trees

Besides points for individual circuits, points are also given for matching sub-trees. The sub-tree can be of depth two or depth three. Matching sub-trees are assigned one point per matching circuit and one point per matching node. The matching sub-trees have to be at least full depth two tree, a full depth two tree is shown in 7.2. Because a matching sub-tree means the distance in the search space between the individuals is much lower than when only individual circuits match. The circuits will be assigned points twice, one time because of the sub-tree and one time because of the individual circuits matching.

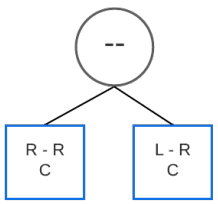


Figure 7.2: Tree of depth 2

7.3. Diversity in cost

Diversity in cost is simple to calculate because the costs of the individuals in a population are known. Variance could be used for the diversity in cost. To compare the diversity in cost with the diversity in structure a slightly different approach is used. The same approach of comparing two individuals as can be read in section 7.4 is used.

7.4. Application of diversity function

Two trees can now be compared for resemblance. The resemblance of a whole population is found in the following way: every individual in the population is compared with 8 other individuals. The average of all these comparisons is then calculated. Because a speaker filter circuit contains three trees, one for each filter, this is done for all bass, mid-toner and tweeter trees. What is calculated with the point system is not really diversity, if the points are lower diversity is lower and if the points are higher the circuits resemble each other more. The term resemblance will therefore be used in the rest of this paper as a metric, resemblance can be regarded as the opposite of diversity.

7.5. Diversity in structure or cost

In order to determine the suitability of diversity in cost or structure as indicator, both were tested on a GA run. The results are shown in the figure 7.3.

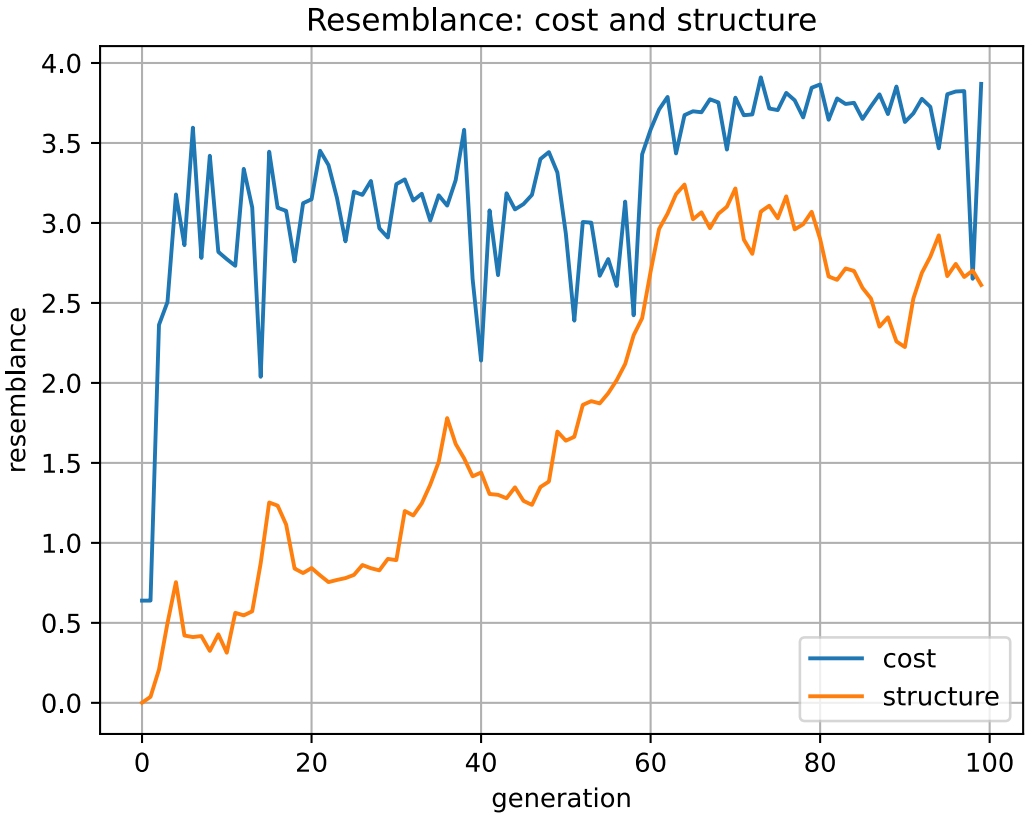


Figure 7.3: Resemblance in cost vs resemblance in structure. 100 Generations, Tournament size 2, Selection size 70

As can be derived from the figure, no clear correlation between resemblance in cost and structure is found. The only relation that can be found is the sudden increase of cost resemblance around generation 60, when the resemblance in structure increases. The absence of a clear correlation can be explained by the following: When the resemblance in structure is high the resemblance in fitness does not need to be high, a change in one component can result in a drastically different fitness even

though the resemblance in structure does not change that much. This may be characteristic for the GA, a small change in the tree can make a huge difference in functionality. GA's for other applications may show a stronger correlation. In the figure it can be seen that the diversity in cost does not show the stage the GA is in, it quickly jumps to around 3 and stays there for most of the generations until it has another jump at generation 60. Because diversity shows in much more detail how the diversity decreases throughout the generations it will be used instead of the diversity in cost. Also a more general hypothesis can be formulated: If a small change in structure in a model results in a large change in fitness, diversity in fitness should not be used in judging the diversity of a population.

8

Methodology

Ideally parameters are tuned based on their outcome: all selection methods and their parameters are tried and the resulting fitness is compared. However, doing so would need a lot of computing power. To give an idea of how long it takes to do one run of the genetic algorithm: running till the cost has fully saturated takes around 4 hours on the HP ZBook Studio G5 with an Intel Core i7-9750H CPU at 2.60 GHz. Tournament selection has three parameters excluding extra options that exist to improve selection. For all values of these parameters the genetic algorithm would have to be run on different speakers and multiple times to give an accurate view on how well the parameters perform. This may be possible for a large company like google with multiple supercomputers but is normally not an option. To counter this computing power problem different parameters are tested for a limited amount of generations: The cost has not fully saturated yet. To still accurately predict the effectiveness of tournament selection parameters the following method was used. A large part of this thesis uses the methodology defined here, but that later in the project the GA was changed in such a way more generations could be run within the same amount of time. This is used for the convergence stadium runs in section 10.2. Also the methodology described here can be used for other slow GA's like GA's that use SPICE to calculate the cost.

8.1. Cost and Diversity

Normally cost is used to evaluate selection quality. But to obtain more information the diversity function will also be applied in the analysis of parameters. Looking at the diversity and cost a few situations can occur:

- The diversity is low but the cost is decreasing: even though the GA is no longer exploring it has not fully exploited a local minimum and the cost can still decrease further.
- The diversity is low and the cost has stagnated: The GA is now in a local minimum and is no longer exploring. The cost will not further decrease because it is stuck in a local minimum and has fully exploited this minimum.
- Diversity is high and the cost has stagnated: The GA is still exploring but can not converge to a minimum.
- Diversity is high and the cost is decreasing: The GA is exploring and finding new minima.

8.2. Method

Looking at these situations a method to tune the parameters is formulated. It is important to note that there is no linear correlation between the diversity and how much potential there is to lower the cost further. Increasing the selection pressure in a high diversity but stagnated cost situation does not always improve the cost further. This could happen when the population has found multiple minima with an almost equal fitness that are fully exploited. This means there is no clear way to decide which parameter value is better if one has a higher cost and the other has a lower diversity. Still, some general

rules can be formulated with regard to the diversity and cost of parameters. These are explained with examples from the tournament size tuning in section 9.5 and the section about convergence 10.2.

- If one parameter value can achieve the same cost with a higher diversity it is better, because it still has potential of decreasing the cost. A tournament size of 2 keeps a low diversity and still keeps decreasing after 70 generations as can be seen in 9.5.
- If the diversity is high but the cost stagnates the selection pressure is not high enough and should be increased. A tournament size of 3 can achieve a lower cost by increasing the selection pressure once the cost has stagnated as can be seen in 10.2.
- If the diversity is low and the cost stagnates at a high value the selection pressure should be decreased. A tournament size of 20 quickly gets a very low diversity and stagnates at 30 as can be seen in 9.5.

9

Tournament Selection

In this section the parameters of tournament selection will be evaluated and tuned to fit the application in the GA. Besides the general structure of the GA there are some parameters involved which specify the mutation and crossover behaviour. The probabilities of different kind of changes to the tree that were chosen by the Mutator group are shown in table 9.1. Resistors, inductors and and capacitors all have an equal chance of being added in a circuit and the tree has a maximum depth of 4. These are the parameters that will be used during the tuning process in this section. Speaker 1 and speaker 2 will be used in the tuning process, the acoustic response of these three-way speakers can be found in 13.2.

9.1. Cost

Diversity in structure and the cost will be used to measure the performance of the selection methods. Cost is a measure for fitness, if the cost is lower the individual is fitter. The genetic algorithm uses two different cost functions as specified by the controller. Cost function 1 is used until the cost drops below 30. After that cost function 1 and 2 will be used in an alternating fashion. Cost function 1 focuses making the drivers operate within their range while cost function 2 focuses on creating a flat acoustic response. Unless stated otherwise, cost function 1 will be used in the tuning because it is the one used until the cost drops below 30. Because the GA is computationally expensive and will not be run for many generations most of the generation in running the GA will optimize this cost.

Change	Probability
Mutate Circuit From List	10%
Mutate Circuit From Tree	5%
Mutate Node From List	5%
Mutate Node From Tree	5%
Remove Node	5%
Add Node	10%
Component Value Change	50%
Cross-Over	10%

Table 9.1: Probability of chances to the tree in the Mutator

9.2. Parameters

For tournament selection method there are three parameters that need to be tuned. Population size (N), selection size (M) and tournament size (K). The meaning of these parameters are elaborated in section 6.4. It is difficult to determine accurately which parameter values are the best to use because performance of the parameters are dependent on each other. When population size is changed the optimal value for selection size and tournament size also change. Running the algorithm for data is time-expensive and it is therefore not possible to just run the algorithm for all value combinations and see which works best. The method used to determine the parameter values is making a good guess for the population size, after that making an educated guess for the selection size and then tuning the tournament size. Making educated guesses is based on the mathematics of tournament selection. Population size was chosen to be first selected because performance is not sensitive to small population size differences and a size that is roughly correct will create a working algorithm, as long as selection size and tournament size are tuned accordingly. It is possible to determine a selection size based on the mathematics of tournament selection and be sure that is will at least perform reasonably well, this is not the case for tournament size. That is why tournament size was chosen as the parameter that is tuned.

9.3. Population Size

Genetic drift is when the population drifts to a certain region in the search space instead of exploring multiple regions at once. A smaller population size has more genetic drift because less solutions are kept alive and in this way the genetic drift is higher because regions with a slightly lower fitness will be eliminated from the population [2]. Article [33] analyses other papers on population size and comes with the following conclusions: with large populations fitness grows more slowly but will find a better end result. The dependency of generations needed before convergence and population size remains unclear. Genetic algorithms with more drastic changes to the population, for example when cross-over can replace big parts of the individual, need a smaller population because the population will stay more spread over the search space instead of gathering in a small region. Different functions have different optimal population sizes. Articles found on the internet show a widely varying optimal population size for different functions so it is it is best to search for good population sizes close to what others have found to be optimal when creating an electric circuit, most have used a population size between 50 and 200. Most papers do not go into detail and the decision on population size seems rather arbitrary. Therefore articles focused on researching population sizes for different functions will also be considered.

Article [14] shows the performance of different population sizes for different functions, one of which considers a tree of 16 components that can each have 4 different values. The trees created for the speaker driver filters are equal in size but the components can have more different values. The paper shows that a population of around 1000 is needed for the algorithm to find a good solution.

9.3.1. Procedure

Now we have a range to search in for population size, namely 50 to 1000. Because of the large search space in generating a speaker filter circuit compared to most other circuit generating genetic algorithms, it is expected a higher population size is needed so the lower limit of 50 will not be tested. A population size of 100, 200 and 500 will be tested for speaker 1, a population size of 1000 was not tested because the population size of 500 already turned out to be less effective than 200. The selection size is set to 30%, which will be justified in the selection size chapter.The tournament size will be set to 5, a size that generally works well for the tournament selection method.

9.3.2. Results

The cost and resemblance of the genetic algorithm per generation for a population of 100, 200 and 500 is shown in figure 9.1. For the 200 and 500 plot some oscillating behaviour can be seen, this is due to the change in cost function once the cost drops below 30, the graph displays the cost function used in the generation. To compare the graphs, the lower parts of the oscillations can be ignored because these correspond to cost function 2 instead of cost function 1. For a population of 100 the average of three runs is plotted while for 200 and 500 only one is plotted. This is due to the significant amount of

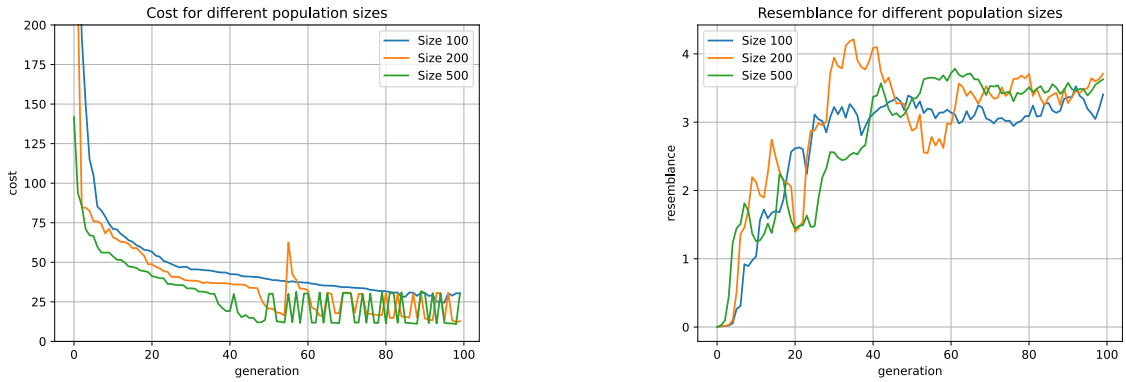


Figure 9.1: Cost and resemblance for different population sizes

time it takes for the genetic algorithm to run with populations of 200 and 500. This is also important to realise when analysing the results. The time it takes the algorithm to finish the same amount of generations is linearly correlated with population size. So the population size of 500 takes five times as long. This means the population size of 100 could be run for 500 generations and maybe create a better circuit in the same amount of time, even though the cost after the same amount of generations is worse.

9.3.3. Conclusion and Discussion

The equal resemblances for different population sizes came as a surprise, a lower resemblance was expected for higher population sizes. This could be explained in the following way: A bigger population size explores more, but there is always a minimum that is clearly better than the other minima. Individuals in other minima are quickly eliminated and only one remains. The fact that the population size of 100 has a lower cost means it has not found this particular minimum that the population sizes of 200 and 500 were able to find. A population size of 100 does not achieve the same cost as the higher population sizes and seems to stagnate. This means it quickly finds a local minimum and is therefore not large enough to explore the search space well. The costs of the population size of 200 and 500 are comparable, as is the resemblance. This means the extra exploration of population size 500 instead of 200 is not needed to find the wanted minimum. The population size of 500 takes 2.5 times as much time to train. Because the there is no improvement in increasing the population size above 200 and a population size of 100 is worse, the population size of 200 is selected for the speaker filter design genetic algorithm.

9.4. Tournament size

Next to the population size, the optimal tournament size must be determined. As already stated in chapter 6.4, the tournament size represents the size of the group from which the fittest individual is chosen. In addition the tournament size directly influences the selection pressure. This becomes clear from the mathematical analysis of tournament selection [3]. The selection probability of an individual with a certain rank based on his score during one tournament can be calculated by the following equation :

$$P_i = N^{-k}((N - i + 1)^k) - (N - i)^k) \tag{9.1}$$

Where i represents the rank of an individual in the population, where 1 is the highest rank and N the lowest. This model assumes that duplicate scores do not occur, which is reasonable, because the score is represented by a floating point number. As shown above, the selection probability relies on the population size (N) and the tournament size (K). For the determined population of 200 individuals, the probability distributions for different tournament sizes are plotted below.

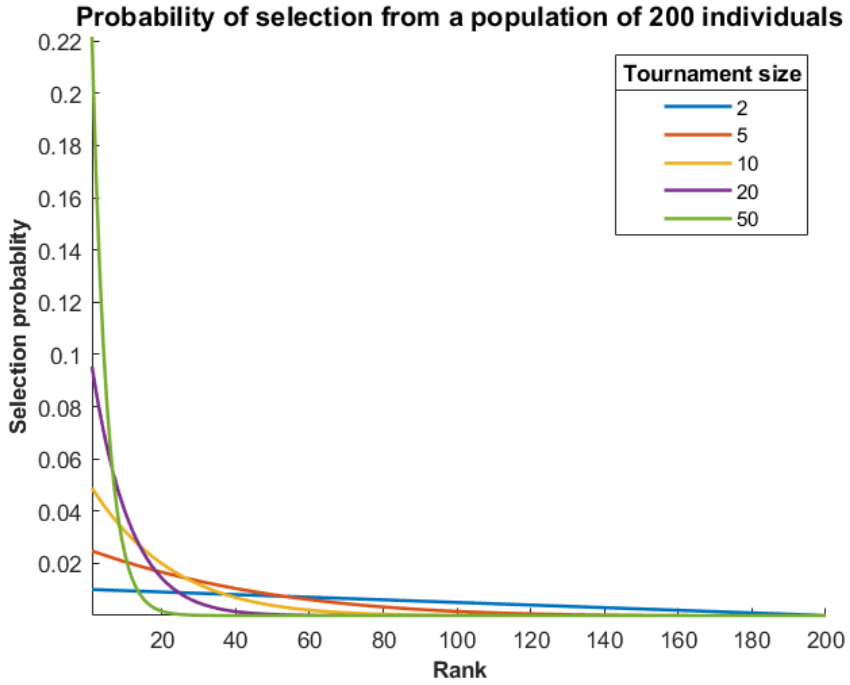


Figure 9.2: Tournament selection method

As expected, it becomes clear from the figure that the probability that the fittest individuals are selected increases with tournament size. When considering a tournament size of 50, the probability that the fittest individual is selected becomes 22%. This means that on average 22% of the population that survived the tournament consist of this individual. In addition, the chance that an individual with a rank of lower than 20 is selected is nihil. This drastically reduces the diversity in the population and increases the chance of premature convergence. On the other hand when a tournament size of 2 is chosen, the probability of selection is more uniformly distributed. This results in a more diverse population which was also found by [10]. However, the best individuals are not prioritized as much with a lower tournament size. In order to realize an adequate balance between prioritizing the fittest individual so the genetic algorithm is able to converge and maintaining diversity, the tournament size will be tuned by trial and error within the range of 2 to 20.

9.5. Selection Size

In addition to the tournament size, the selection size must be chosen. The selection size is equal to how many tournaments are performed. As specified by the mutation group, all the children that survive the tournament are automatically part of the children of the next generation. The new population contains the selected children and the rest of the population is filled with mutated and crossovered variants of the selected children. There are two factors that need to be taken into account in order to determine the suitable amount of tournaments. This will be demonstrated by the following two extreme scenario's.
Scenario 1: When the amount of tournaments is equal to the population size, the whole population is filled with selected children. In this case no mutations and no crossover occurs. The search space is not explored anymore. Eventually the population will consist only of the best individual. This is useless, because no solutions other than the initial population will be explored.
Scenario 2: In each generation only one tournament occurs. In this case only K individuals are considered in the selection procedure. The new population now consists of the one individual that won the tournament and N-1 crossovered and mutated versions of itself. In this case a large part of the initial population is not even considered for selection. Since the mutation and crossover step are computationally intensive, this is a waste resources. In addition, there is a high chance that fit individuals remain unnoticed. Besides that, only the search space near one particular solution is investigated.
Based on these two scenarios it becomes clear that the amount of tournaments influences the exploitative and exploratory behaviour of the algorithm. In figure 9.3 the product of the tournament size and

selection size is plotted against the expected number of individuals that are not sampled in any tournament,according to equation [36]. A population of 200 individuals is chosen in accordance with section 9.3.

$$N \cdot (\frac{N}{N-1})^{-KM} \tag{9.2}$$

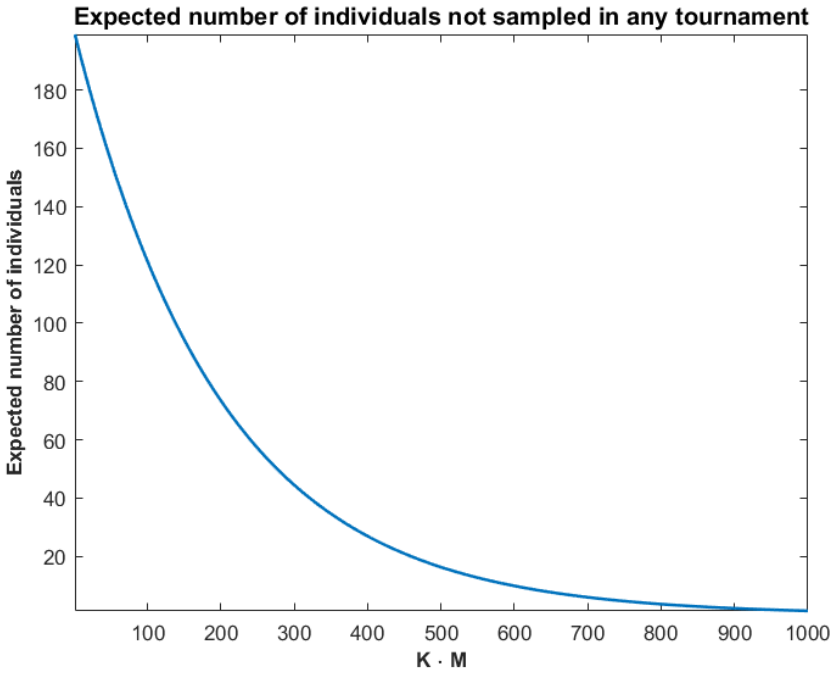


Figure 9.3: Individuals not samples with respect to selection size times tournament size

As shown in the graph above, a larger product of tournament size and selection size results on average in more individuals involved in the selection process. In order to minimise the loss of computational power and the loss of fit individuals, only selection sizes where at least 50% of the population are involved in the selection process will be considered. This corresponds to $K \cdot M \geq 140$ In order to explore new solutions near all the selected individuals, the selection size should be at most 100 to form at least one mutated or crossovered individual per child. As determined in section 9.4, only tournament sizes in the range of 2 to 20 will be considered. In order for all tournament sizes to meet the requirements above, a selection size of 70 is chosen.

9.5.1. Procedure

Based on the selected population size of 200 and selection size of 70, the tournament size is empirically tuned in the the range 2 until 20. This section describes the procedure that was followed during the tuning process.

In order to determine the suitable tournament size, the genetic algorithm is ran on different tournament sizes for two different speakers. The following tournament sizes are analysed: 2, 3, 4, 5, 10, and 20. The genetic algorithm is ran three times per tournament size to create an average which is compared. The amount of generations was limited to 75 because the cost has practically stabilized at that point for all tournament sizes, but some still have a slight slope. Based on the measures of cost and resemblance and how they vary over the generations the appropriate tournament size can be selected.

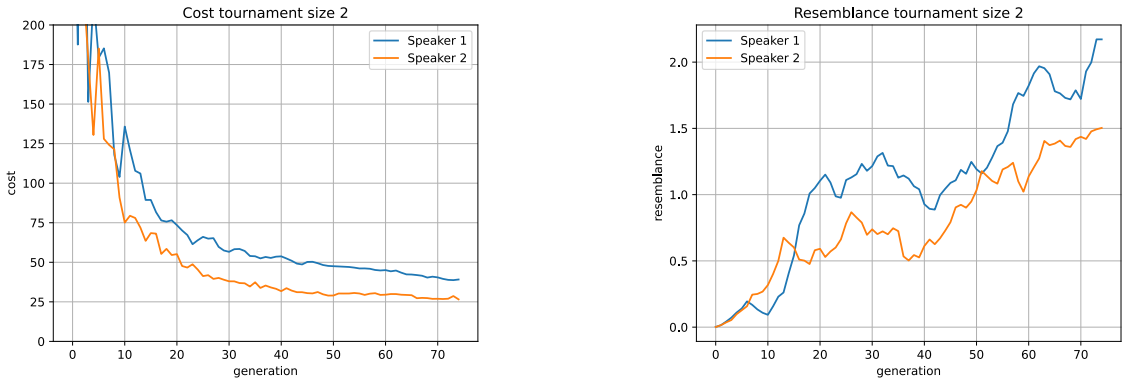


Figure 9.4: Cost and resemblance of 2 speakers with tournament size 2

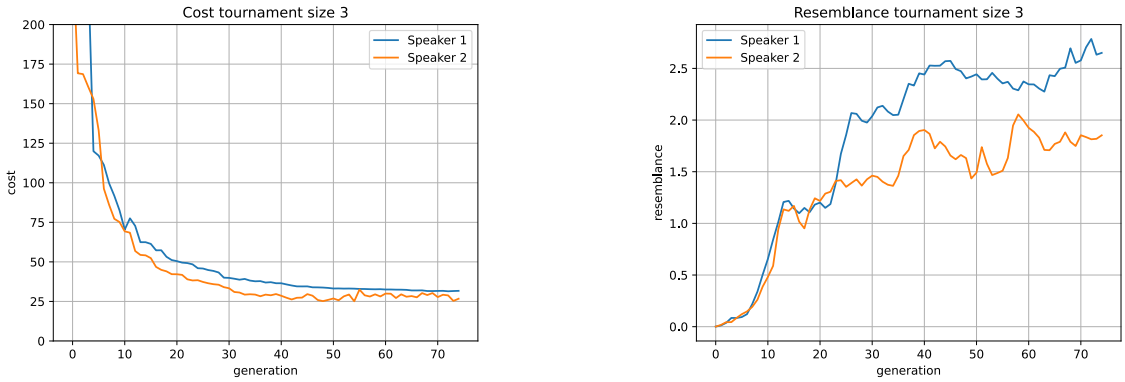


Figure 9.5: Cost and resemblance of 2 speakers with tournament size 3

9.5.2. Results

The results of the three individual runs per tournament size for both speakers can be found in the appendix 13.3. The average of these 3 runs per tournament size are displayed in the figures 9.4, 9.4, 9.5, 9.6, 9.7, 9.8 and 9.9. Note that the scale of the y-axis on the resemblance figures is not equal for all tournament sizes, this is done so the diversity of the two speakers can be compared more accurately.

9.5.3. Conclusion and discussion

Above the minimum cost and resemblance per generation are plotted. At first glance it becomes clear that the resemblance increases with the tournament size. This is as expected and in accordance with figure 9.3. As the tournament size increases the percentage of the fittest individuals in the population increases. As a result the population becomes less diverse. However the increase in resemblance is getting smaller with increasing tournament size. This is also as expected because the probability distributions are quite similar for high tournament sizes as can be seen in figure 9.3. What is remarkable is that the costs for different tournament sizes are close to each other after 75 generations. Every tournament size of 3 and higher has a cost of around 30 at generation 75. Only the tournament size two has a slightly higher cost of around 35. Because diversity is an indicator for how much an algorithm is exploring, it is best to choose a tournament size that achieves a low cost and has a high diversity, so there is potential to get an even lower cost. When considering the cost and resemblance plots for the different tournament sizes for speaker 2, it becomes clear that a tournament size of two provides the best performance. All the cost graphs saturate at 25. However given this cost, a tournament size of 2 provides the lowest resemblance of 1.5. This conclusion is similar when considering speaker one. In this case almost all tournament sizes end with a cost of around 30, with tournament size 2 having a slightly higher cost. The resemblance at generation 75 is for tournament size 2 the lowest with a value of 1.7. For this reasons a tournament size of two is chosen.

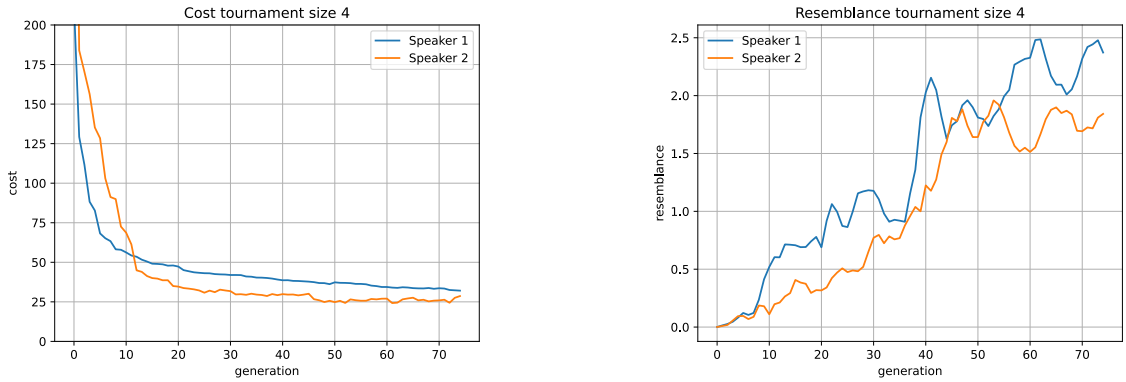


Figure 9.6: Cost and resemblance of 2 speakers with tournament size 4

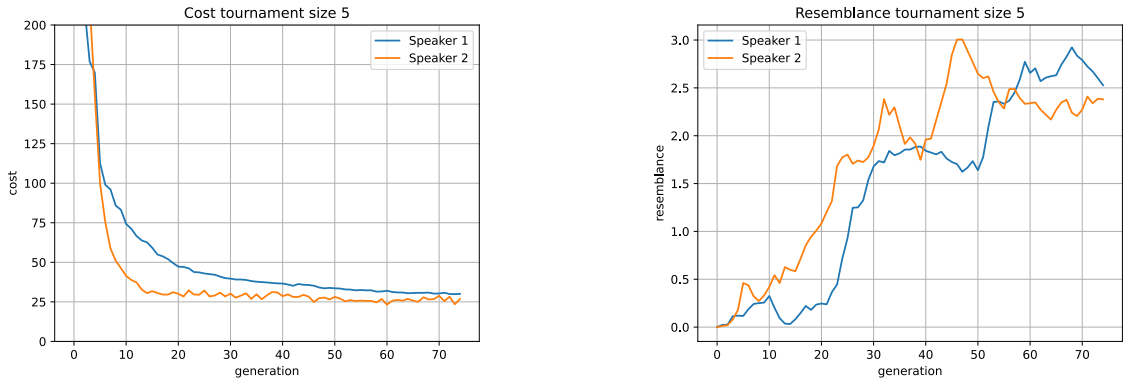


Figure 9.7: Cost and resemblance of 2 speakers with tournament size 5

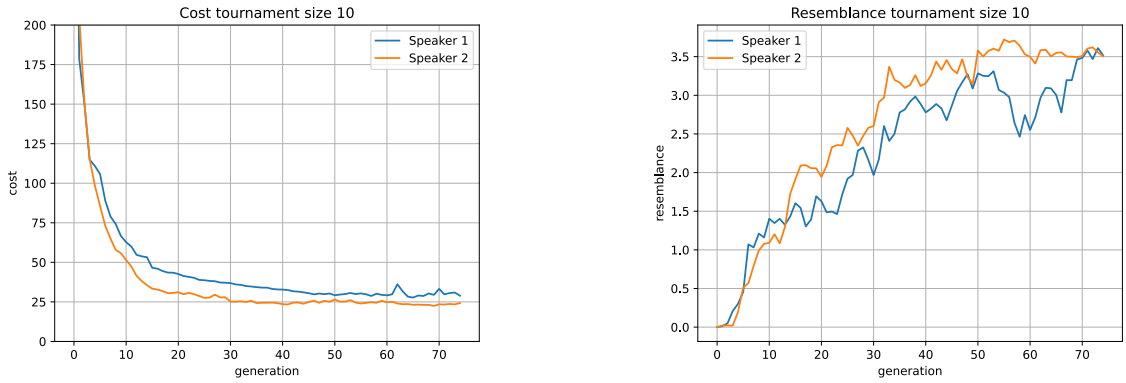


Figure 9.8: Cost and resemblance of 2 speakers with tournament size 10

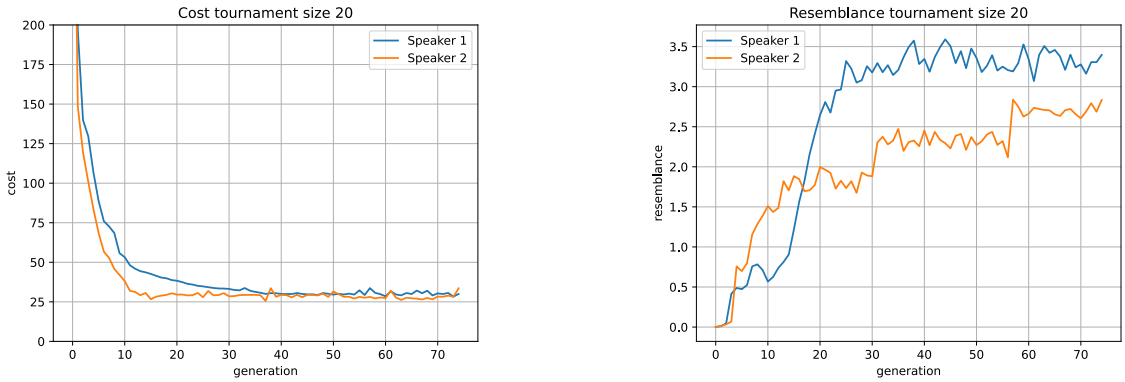


Figure 9.9: Cost and resemblance of 2 speakers with tournament size 20

9.6. New tournament size data

During the last week of the project the speed of the GA had been improved. This made it possible to run the GA for many generations. Finally all tournament sizes had been run for 400 generations, because at this generation the cost decline slowed down and did not change much more. The costs for tournament size 2, 3, 4, 5 and 7 were compared at generation 400. Cost function 2 was used, because that function is optimized when more generations are run. Again, for every tournament size 3 runs were done and the average was used to compare the different tournament sizes.

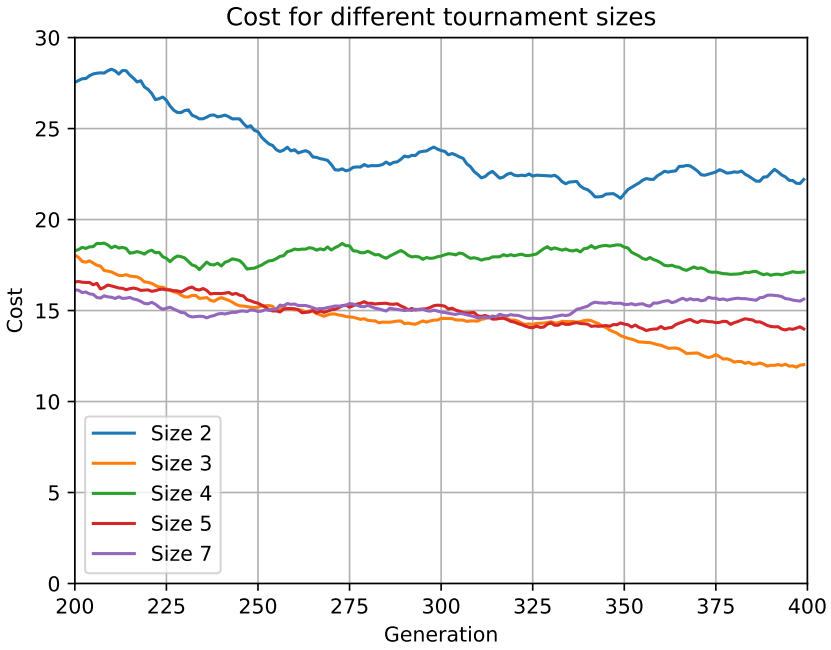


Figure 9.10: Resulting cost per tournament size

9.6.1. Result

The results are shown in figure 9.10. The figure shows an exponential moving average (EMA) with a multiplier of 0.1. The moving average was taken because the costs had a lot of noise. The formula for the exponential average is shown in equation 9.3.

$$EMA[i] = cost[i] \cdot multiplier + EMA[i-1] \cdot (1 - multiplier) \tag{9.3}$$

9.6.2. Conclusion and Discussion

From figure 9.10 it can be concluded that a tournament size of 3 achieves the lowest cost. This is unexpected because based on the resemblance and cost it was earlier predicted that a tournament size of 2 would produce the best result. This could be caused by the fact that a GA with tournament size 2 is unable to converge because of the low selection pressure. Section 10.2 investigates if this is the case by increasing the selection pressure when the cost stagnates.

10

Adaptive selection operators

10.1. Adaptive selection using noise

When considering tournament selection, the magnitude of the difference in fitness between individuals is not taken into account. The selection probability is solely based on the rank of the individual. The selection pressure is also constant due to the fixed tournament size. Due to the dynamic properties of the genetic algorithm it makes more sense to adapt the selection pressure based on the dynamic behaviour of the algorithm. In this section a modified version of tournament selection is implemented, which is inspired by this paper [28], which suggest a method to adapt the selection pressure based on the dynamic properties of the GA. As stated in section 6.4, tournament selection consist of two stages, a sampling stage and a selection stage. In this adapted version, an extra stage between the sampling and selection stage is added. In this stage noise is added to the fitness values of the sampled individuals. This noise is proportional to the difference in fitness values. Based on the formula 10.1 a new score is calculated:

$$Score = Fitness + (Fitness_{max} - Fitness_{average}) \cdot X \tag{10.1}$$

Where X is a random variable with a continuous uniform distribution, with the probability density function shown in equation 10.2.

$$f(x) = \begin{cases} 1 \; for \; 0 \leq x \leq 1 \\ 0 \; for \; x < 0 \; or \; x > 1 \end{cases} \tag{10.2}$$

The idea behind this score is to adjust the selection pressure during the generations of the GA. In the first few populations of the GA the average fitness is quite low, except for some super individuals, who have a high fitness. Due to the addition of noise, the dominance of the super individuals is decreased. However, they still on average have a higher change to be selected, because the noise is added to the original fitness. In addition, the selection probability of less fit individuals is increased due to the addition of the noise. This results in a lower selection pressure during the first few generations.
When the population starts to converge to a minimum, the variance in fitness becomes smaller. In this case the amount of added noise is smaller and the scores are almost the same as the fitness. The actual fitness values are now considered in the selection, which results in an increased selection pressure, which speeds up the convergence stage.

10.1.1. Procedure

When adaptive tournament selection is used, it is likely that the optimal tournament size changes. This is because the noise may allow higher tournament sizes to retain a higher diversity and it may make lower tournament sizes unable to converge. Therefore the tournament size will be tuned. Adaptive tournament selection is run three times per tournament size for tournament size 3, 5 and 10. The average of these 3 runs is compared to the averages without noise, the measurements without noise have only run for 75 generations. The individual 3 runs per tournament size with noise can be found in the appendix 13.6.

10.1.2. Results

The results are shown in figures 10.1, 10.2 and 10.3.

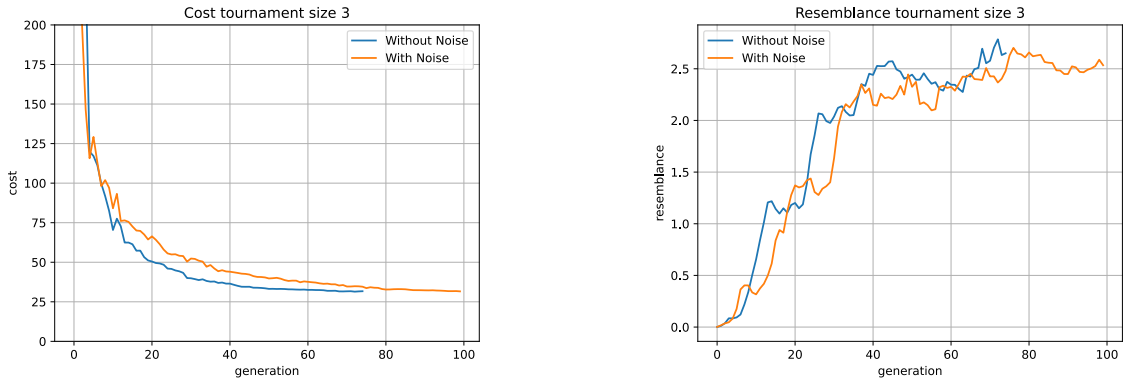


Figure 10.1: Cost and resemblance of 3 runs with tournament size 3 + noise

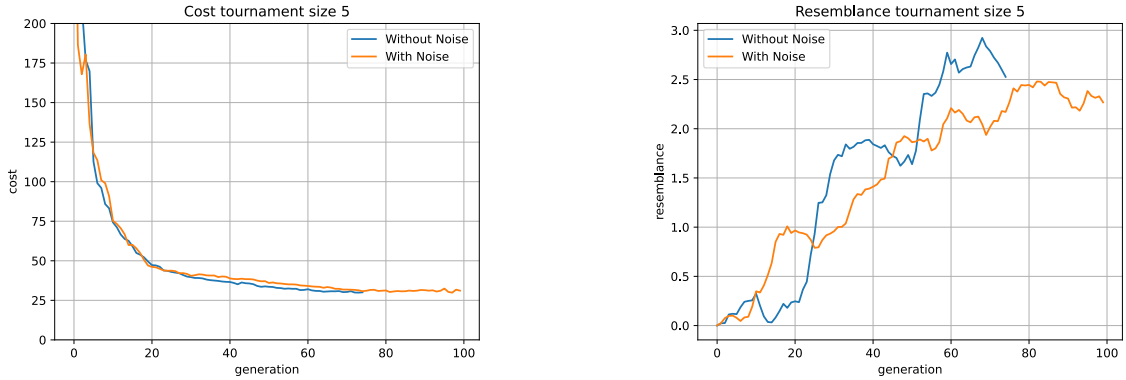


Figure 10.2: Cost and resemblance of 3 runs with tournament size 5 + noise

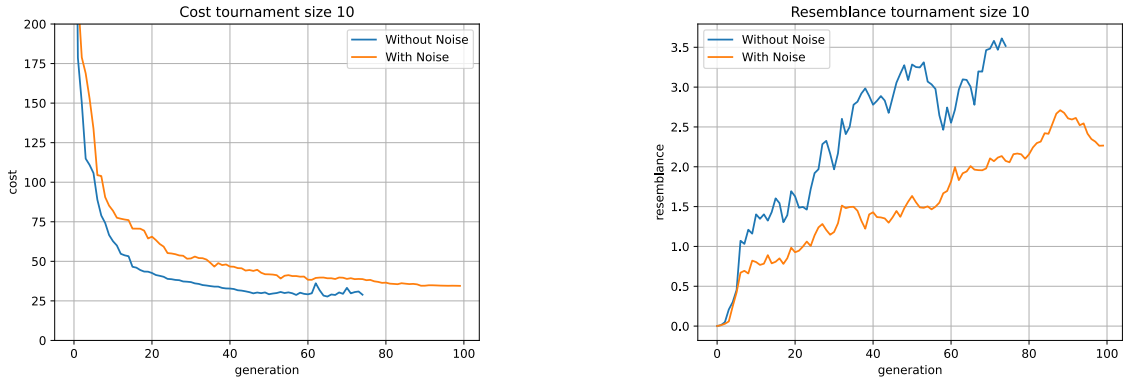


Figure 10.3: Cost and resemblance of 3 runs with tournament size 10 + noise

10.1.3. Conclusion and Discussion

The plot for tournament size 3 with noise 10.1 does not seem to differ much from the plot without noise 9.5, 9.7. This is as expected because this tournament size already has a low selection pressure, so the effect of noise is less clearly visible. A tournament size of 5 performs slightly better with noise than without because it reaches the same cost at the end with a lower resemblance, but the improvement is not significant and could be caused by chance. For a tournament size of 10 it is clearly visible that for the same cost a lower resemblance can be achieved which means the circuit has more potential to find a lower cost after many generations. Even though more data is needed to reach a definitive conclusion, a higher tournament size seems to benefit greatly from added noise to prevent the premature convergence it is normally prone to. Because the noise does not seem to be very influential for lower tournament sizes it will not be implemented, because a tournament size of 2 was found to be optimal in the tournament size section 9.5.3 and the tournament size 10 with noise does not perform better than basic tournament size 2. In addition to the better performance during the first 75 generations, the tournament size with noise will most likely perform as good as without noise during the convergence stage of the GA. In this stadium the variance in fitness becomes smaller and the selection operator resembles normal tournament selection.

10.2. Adaptive selection using convergence stage

Another adaptive selection procedure that could be added is a convergence stage. This is not something done before in the literature, but seems to be a good addition for the following reason: in the results of tournament size tuning it can be seen that lower tournament sizes, so a lower selection pressure, stagnate in cost while keeping a high diversity. This may be because the GA has found some good regions in the search space, but can not fully exploit the minima in these regions because the selection pressure is too low. This could indicate some improvements can still be made by increasing the selection pressure once the cost has stagnated to stop exploring and start the exploitation phase to find the minimum.

10.2.1. Solution

The selection pressure should be dependent on the stage the algorithm is in. When the diversity is high and fitness is low the algorithm is still in its exploring stage. When the diversity is low and fitness is high the individuals are near a minimum in the search space. In this stadium the search space is explored enough and it is then justified to increase the selection pressure in order to converge to a minimum. To function optimally the algorithm should only start increasing the selection pressure when the exploring no longer decreases the cost. Here a new method of increasing selection pressure to converge is proposed.

The selection pressure can be increased by increasing the tournament size. A gradual increase function in tournament size was chosen to ensure a gradual increase in selection pressure. When the algorithm enters the exploitation phase it has a population with a relatively low distance from each other in the search space. Even though the population has a low distance, there are still multiple minima in the area it covers. To find the best local minimum from the minima the selection pressure should be gradually increased. If the selection pressure is immediately set to a high value to converge it will quickly converge to the local minimum close to the individual with currently the lowest cost, even though other individuals might be close to a minimum that is even lower. To see if this theory is indeed true two selection pressure increase methods will be tried, one with a gradual increase and one with an abrupt increase. In this section cost function 2 will be used because of the high amount of generations run.

10.2.2. Procedure

Two selection pressure increase methods will be experimented with. The step function and the stagnation function. All figures contained a lot of noise due to the many generations run, so for all data the exponential moving average with multiplier 0.1 according to equation 9.3 was used. Every experiment shown is the average of 3 runs with the same parameters except for the tournament size 3 with a step function, which is only run once.
Step function: When the convergence stage is started the tournament size is immediately increased from 2 to a tournament size of 3, 4 or 5 and is not changed during the convergence stage. The conver-

gence stage is started at 300 generations and run for 100 generations. The tests are done for a base tournament size of 2 because it was determined it was the best size in section 9.5.3.
Stagnation function: When the convergence stage is started the tournament size is increased by one when the cost stagnates. The cost is seen as stagnated when it has not found a new minimum in the past 80 generations. The number 80 is chosen because for that value the cost has stagnated a few times by generation 3000, which is a manageable amount of generations in terms of run-time. An even higher number of generations will probably be better because it is possible the the cost has not fully stagnated if it has not found a new minimum within the last 80 generations. The GA is run with a starting tournament size of 2 and 3 without stagnation function and 2 and 3 with stagnation function.
Validation: To see which of the stagnation functions should be used the tournament size 2 and 3 with step and stagnation function are run for 3000 generations and the result is compared. 3000 generations was chosen because both convergence stage function have stagnated enough at that point to draw conclusions.

10.2.3. Results

The result of the step function can be seen in figure 10.4 At 300 generations the cost has not fully stagnated, which may influence the results. The results of the stagnation function can be seen in figure 10.5. In all runs of the stagnation function of a starting tournaments size 2 the final tournament size was 4. The first stagnation generation varied widely, from 300 to 1100. For a starting tournament size 3 the final tournament size was 5, 6 and 8. The first stagnation generation was around generation 1300. The compared result of both convergence functions for tournament size 2 can be seen in figure 10.6.

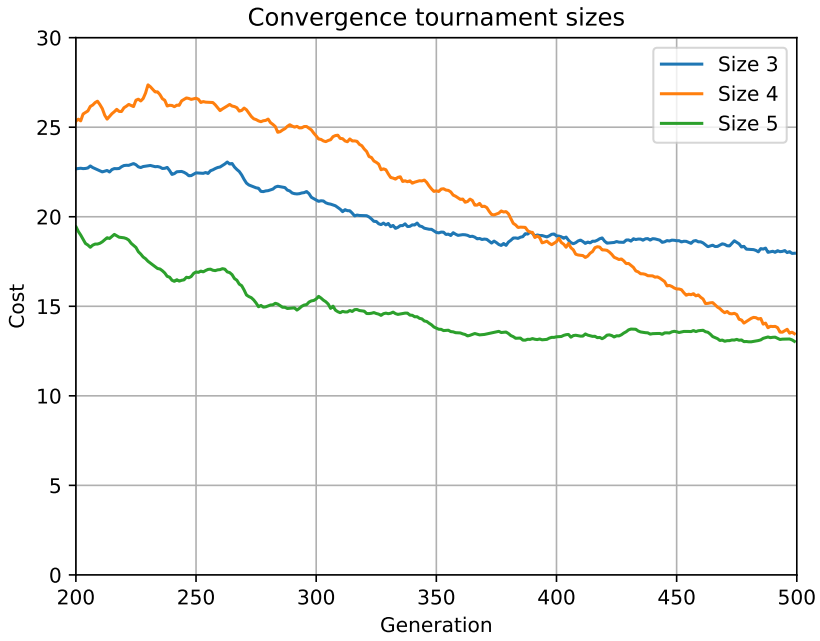


Figure 10.4: Step convergence stage for tournament size 2 to different tournament sizes

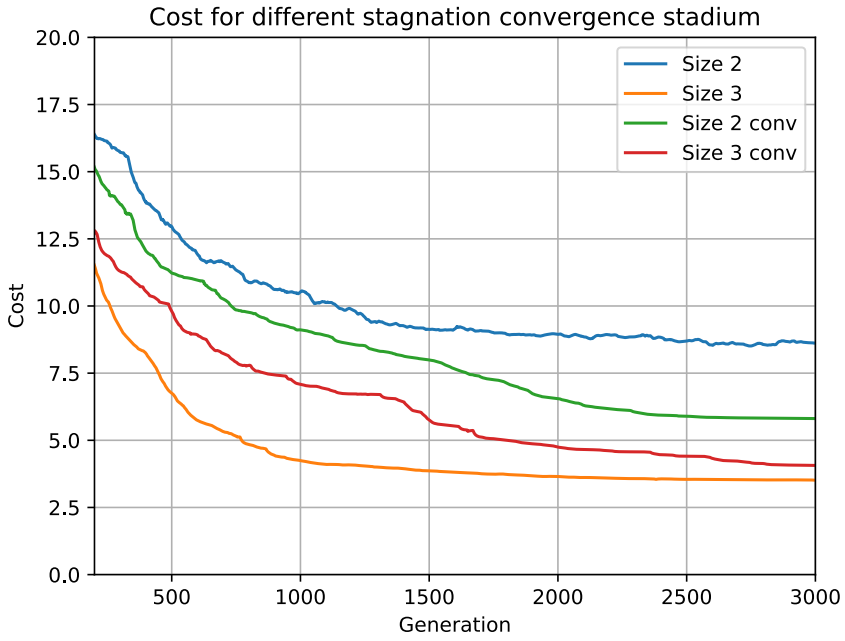


Figure 10.5: Stagnation convergence stage with different tournament sizes

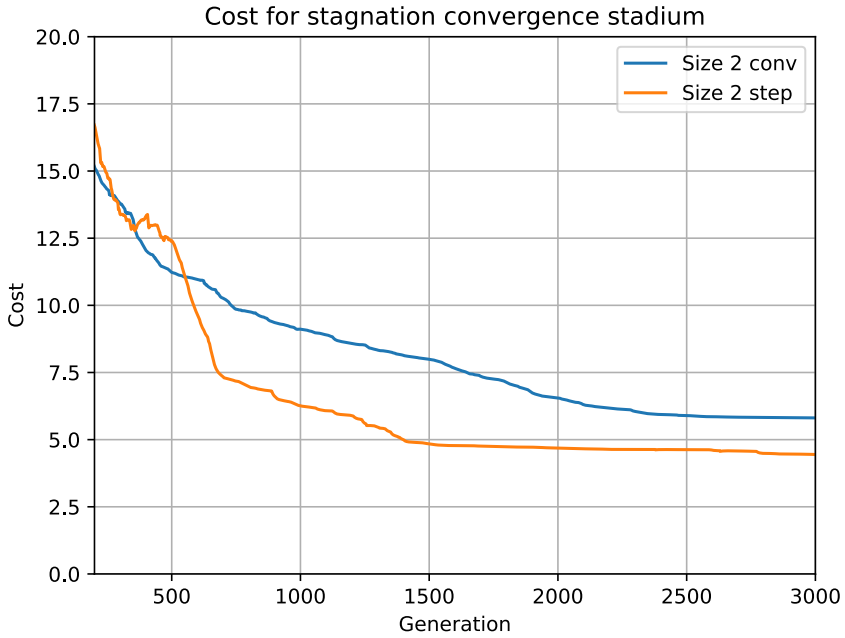


Figure 10.6: Comparison of stagnation and step convergence function

10.2.4. Conclusion and Discussion

In the figure of the step function it can be seen that increasing the tournament size to 3 does not have much effect. Probably because the selection pressure is still too low. Tournament size 4 makes the cost decrease again and even seems to continue decreasing at generation 500 when the run stops, but a clear conclusion can only be taken if the GA has been run for more generations. It was expected tournament size 5 would decrease the cost more rapidly than size 4 but also stagnate earlier, but instead

it does not have much effect on the cost. A possible explanation would be that the selection pressure becomes too high so it quickly finds a bad local minimum. This is however unlikely as a tournament size of 5 is still low. It could also be because the cost of size 5 is already much lower than size 4 and can thus not decrease the cost as much. For now a tournament size of 4 will be chosen as best option because it still has a slope at generation 500 but more test runs could be done to draw a better conclusion.
In the figure of the stagnation function it can be seen that for a tournament size of 2 the stagnation function improves the performance. But for a tournament size of 3 the function has a higher end cost. The first stagnation, and so the increase of tournament size to 4, was after around 1300 generations so the run of the GA for the stagnation function already performed worse before it was activated. After the convergence stage started it can be seen the cost quickly drops where it had stagnated before but it is unsure if the cost could have declined further on its own. For a run of 3000 generations it is concluded the convergence stage for 80 generations with tournament size 3 is the best option.
In figure of the combined stagnation and step function the effect of the stagnation function is shown compared to the step function from tournaments size 2 to 4. Because the run is now done for 3000 generation the step function will be activated after generation 500 so it has a longer exploration time. The step function seems to perform better than the stagnation function. This is surprising and no explanation was found as to why. Apparently the gradual increase is worse than an abrupt increase. For tournament size 3 a step function to 5 was chosen because an increase of 2 was found to be the most effective as step function for tournament size 2. The result can be seen in figure 10.7. Even though the cost at 500 was higher than the normal size 3 tournament selection it quickly decreased until it was lower. It also reached a much better cost at the end than the stagnation function. In conclusion. Tournament size 3 with a step convergence stage is the best option. This is very interesting to see because an abrupt increase in selection pressure is found nowhere in the literature, but the experiments documented here show it is the better option.

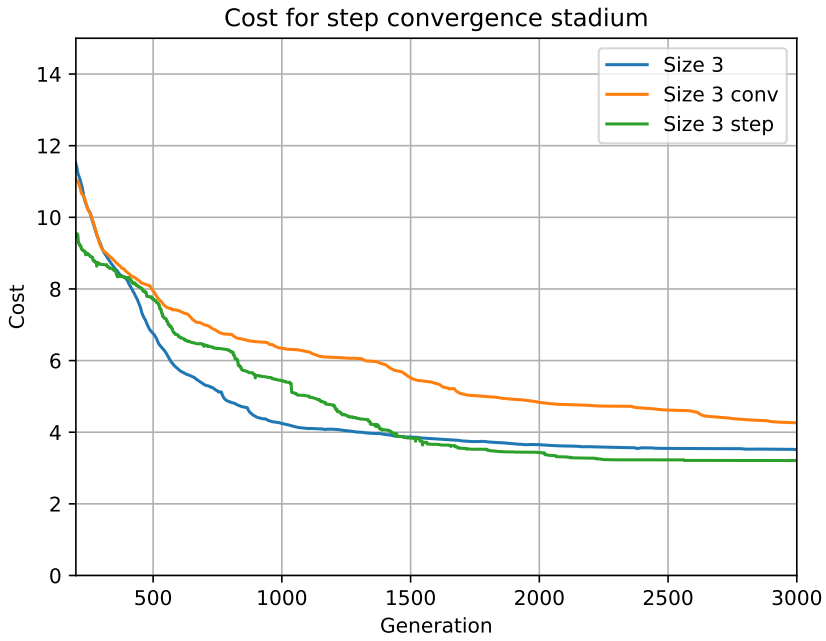


Figure 10.7: Comparison of different convergence stages for tournament size 3

11

Conclusion & future work

11.1. Conclusion

Using the proposed design methodology and the mathematical basis of selection operators, a selection operator for the GA has been chosen. The selection operator enables the GA to create a loudspeaker filter that complies with the project requirements. Tournament selection was found to be the most suitable selection operator. A population size of 200, a selection size of 70 and a tournament size of 3 were chosen as optimal parameter values. For the adaptive selection methods noise was found to be unhelpful, but a convergence stage could improve the functionality of the GA. During the experiments a cost of 3.2 was achieved by tournament size 3 with a step function. This shows that the optimization of the parameters of tournament selection greatly contributed to an improvement in the functionality of the GA and thus the end result. The code for the final selection method can be found in the appendix 13.5. All mandatory requirements were achieved during the project.

11.2. Future Work

The current selection method that is tested seems to be the best option for the GA for speaker filter design. The choice of selection operator is already well researched but literature on which parameters to choose is lacking. Work could be done to make more general rules on which parameters to use based on the application. Also the method for parameters selection based on cost and diversity in early stages proposed in this report is not fully developed yet and further research could be a great addition. A mathematical correlation between diversity and potential to reduce cost further could be created. This way quantitative instead of qualitative conclusions can be made about the performance of a parameter based on the cost and divergence in an early stage. Now divergence is merely decisive when a clear conclusion based on cost can not be made. Also a few other suggestions about future work can be made:

1. Tests done could be done on more speakers. It is possible different tournament selection parameters work better for different speakers. The two speakers that are evaluated in this report are far from enough evidence the optimal parameters found can generally be used as a setup to obtain the best result.

2. Tournament size can be dynamic based on diversity. [25] introduced a dynamic tournament size based on diversity and fitness that shows statistically significant results compared to a basic GA.

3. Some results may be more trustworthy if they are run on the same population. For example, in comparing convergence stages the cost at the start of the convergence stage differs between runs which makes comparing the results hard in some cases. Giving every run the same population at generation 300 would reduce the effect of randomness on the results.

12

Glossary

Here frequently used words or abbreviations that may be hard to understand or unknown are explained.

- Cost: A value assigned to an individual by the cost function. The GA should minimize the cost function.
- Diversity: The difference between individuals in the population: distance between individuals in the search space.
- Fitness: How low the cost of an individual is
- Genetic Algorithm (GA): The AI type used.
- Minimum: A place in the search space where the cost is minimal.
- Minima: Plural of minimum.
- Mutator: The part of the GA that does the mutation, cross-over and other changes to create the new population based on the selected parents.
- Population Size: The amount of individuals created by the mutator from the selected parents.
- Resemblance: Opposite of diversity, individuals with a matching structure have a high resemblance.
- Search Space: All possible filters that can be created as explained in the appendix .
- Selection size: The number of individuals chosen from the population to be parents for the next generation. Equal to the amount of tournaments in tournament selection.
- Tournament Size: The amount of individuals within one tournament where the individual with the highest fitness is selected.

13

Appendix

13.1. Search Space

The task of the GA is function optimization. The input of the function is the filter circuit with all its components and component values and the output is a value, also named cost, for how far the filter is from the requirements. This value should be minimized. The search space, as shown in figure 13.1 is the space of all input values with their cost and in this search space the global minimum should be found, this is the point in the search space with the lowest cost. The search space also has local minima, these are points in the search space where the GA can get stuck.

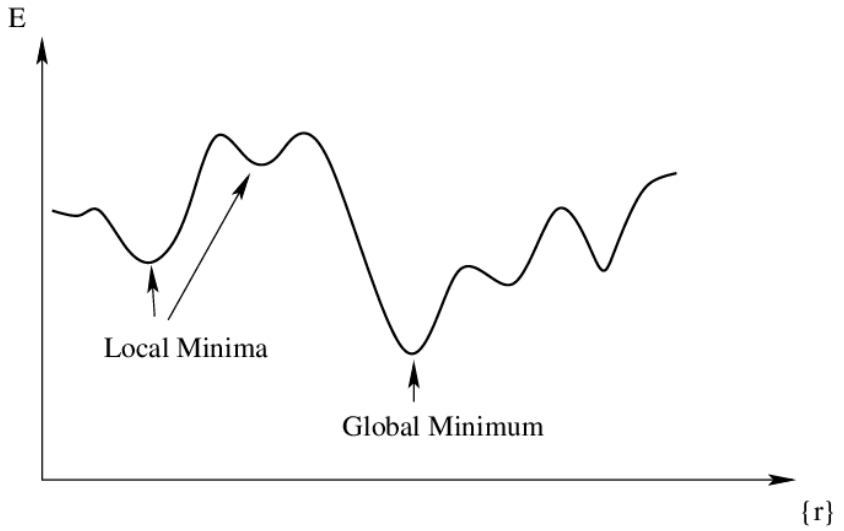


Figure 13.1: Search Space 1D, E is the cost and r the location. Credits to https://vitalflux.com/local-global-maxima-minima-explained-examples/

13.2. Speakers

The acoustic response of speaker 1 can be seen in figure 13.2 and the acoustic response of speaker 2 can be seen in 13.3. Speaker 1 is an unknown model. Speaker 2 is a composition of a woofer (DC130A-8), mid-range (PA165-8) and tweeter (DSN25F-4) from Daytonaudio.com.

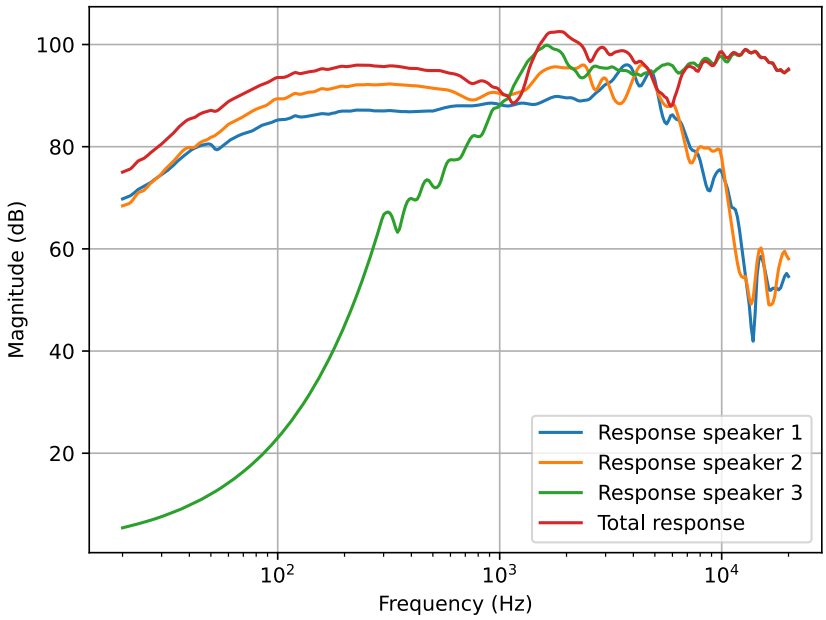


Figure 13.2: Speaker 1 acoustic response

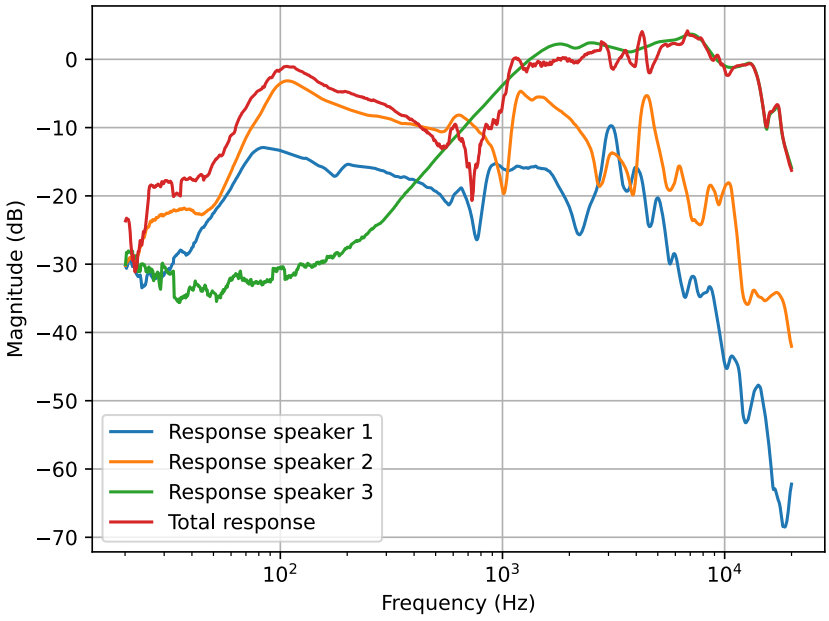


Figure 13.3: Speaker 2 acoustic response

13.3. Figures Tournament Size

The 3 runs per tournament size for sizes of 2, 3, 4, 5, 7, 10 and 20 for speaker 1 can be seen here: 13.4, 13.5, 13.6, 13.6, 13.7, 13.8, 13.9, 13.10. For speaker 2 the runs can be seen here: 13.11, 13.12, 13.13. For speaker 2 the individual run data of tournament sizes 5, 7 and 10 has sadly been lost. The average of the 3 runs of speaker 1 along with the average of speaker 2 is used to decide which

tournament size is optimal in section 9.5.

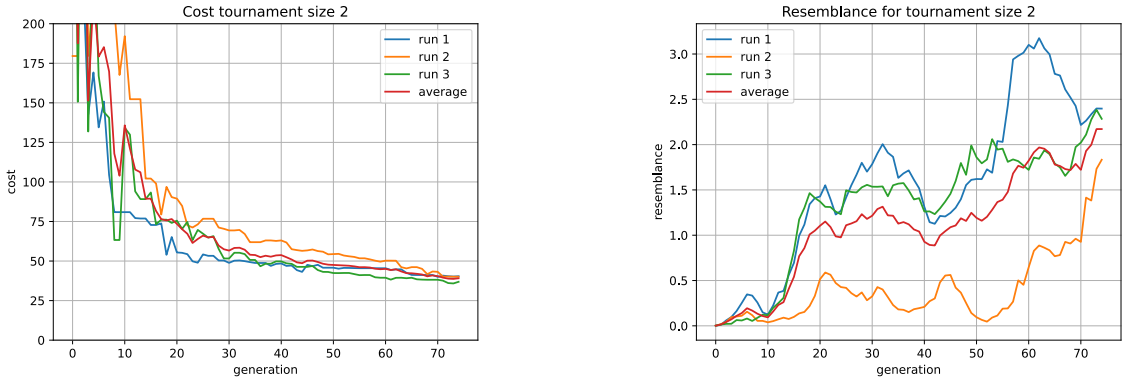


Figure 13.4: Cost and resemblance of 3 runs with tournament size 2

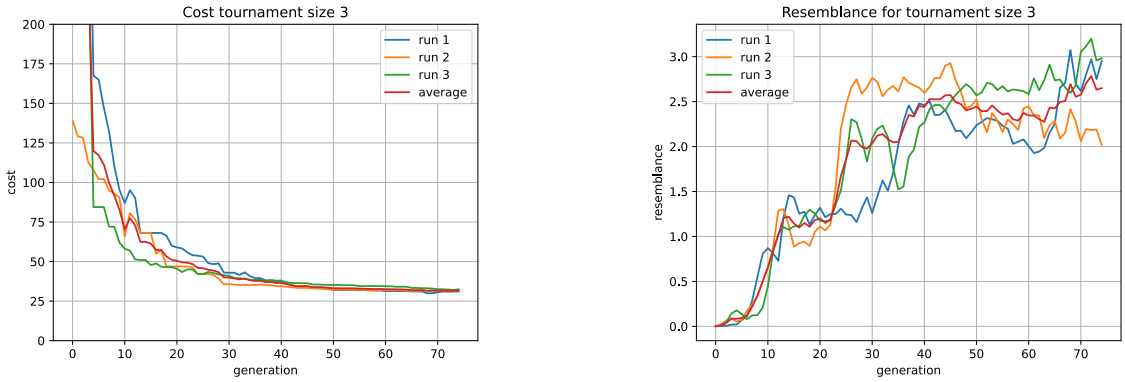


Figure 13.5: Cost and resemblance of 3 runs with tournament size 3

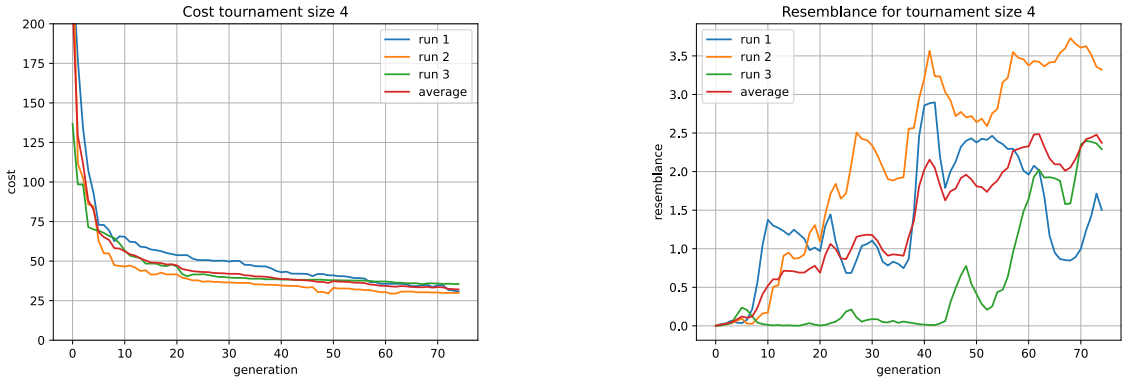


Figure 13.6: Cost and resemblance of 3 runs with tournament size 4

13.4. Diversity Code

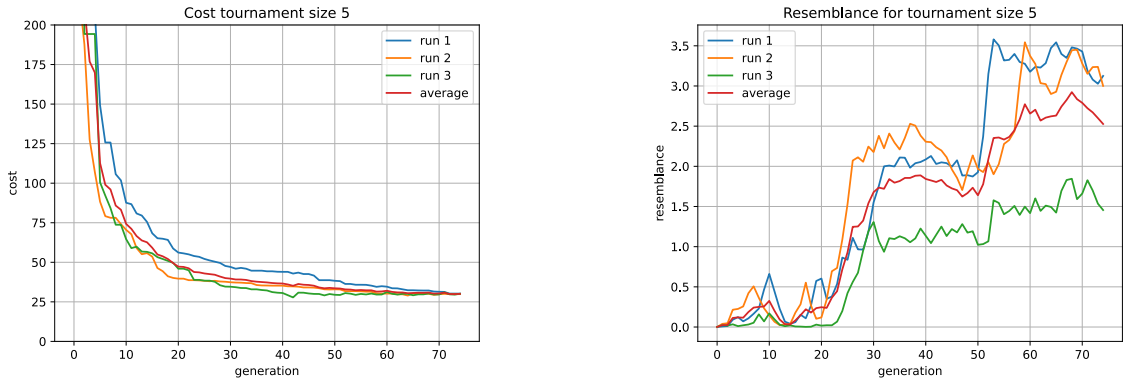


Figure 13.7: Cost and resemblance of 3 runs with tournament size 5

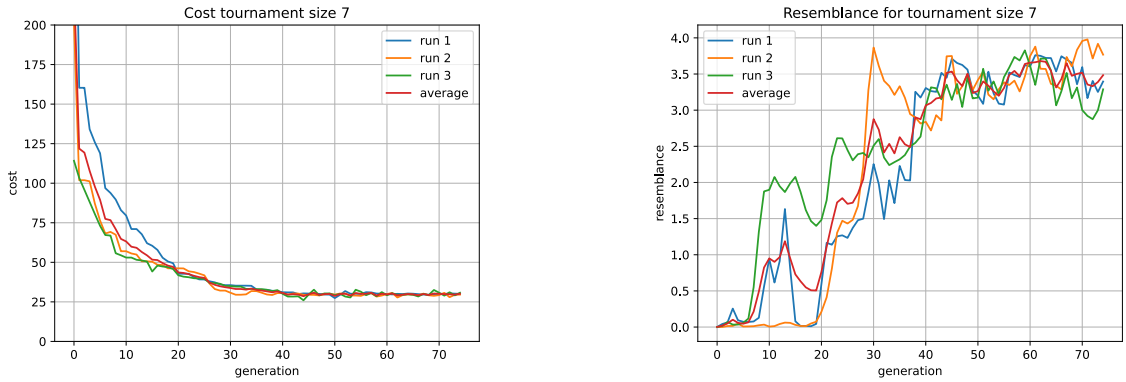


Figure 13.8: Cost and resemblance of 3 runs with tournament size 7

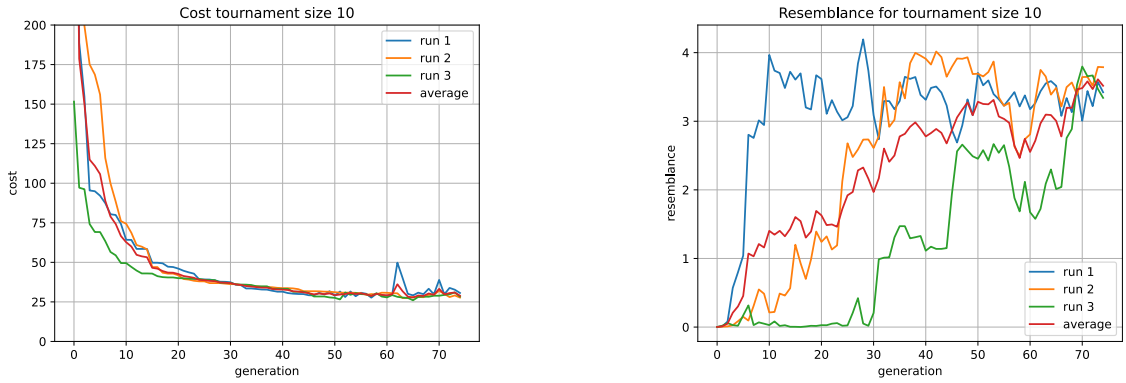
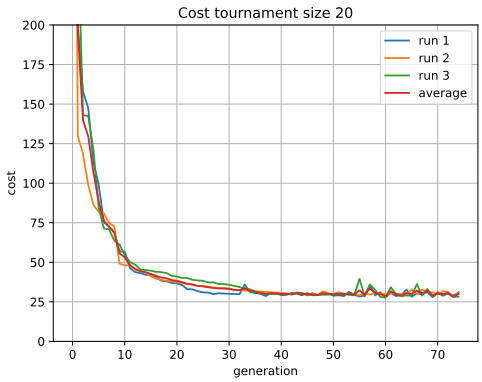


Figure 13.9: Cost and resemblance of 3 runs with tournament size 10



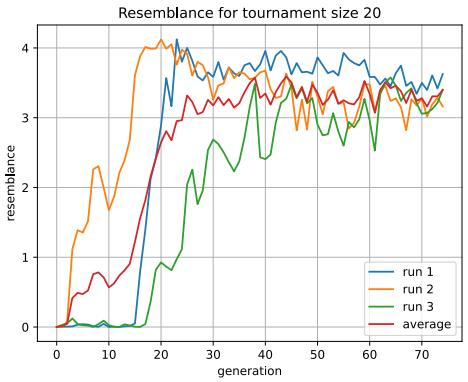
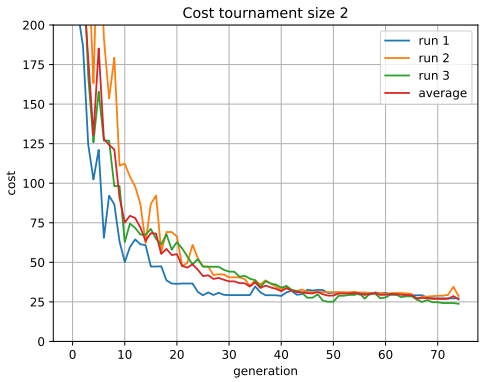


Figure 13.10: Cost and resemblance of 3 runs with tournament size 20



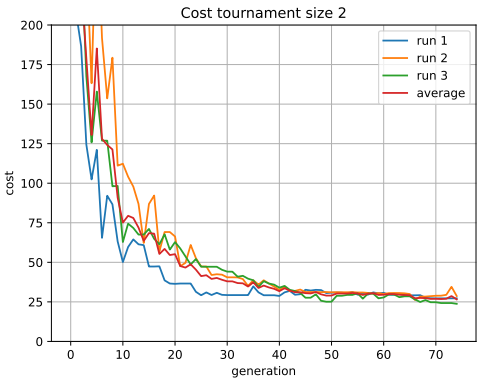
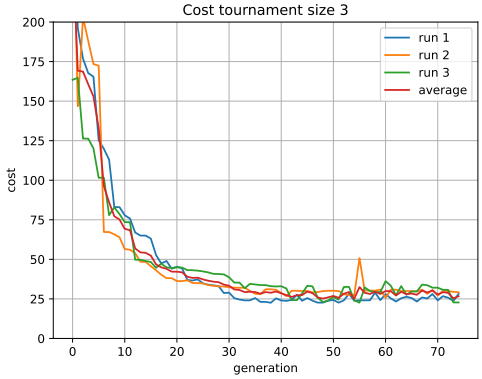


Figure 13.11: Cost and resemblance of 3 runs with tournament size 2



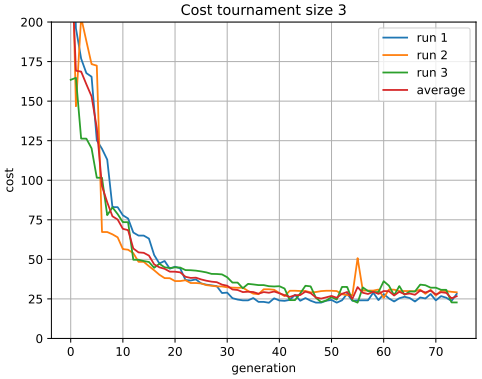


Figure 13.12: Cost and resemblance of 3 runs with tournament size 3

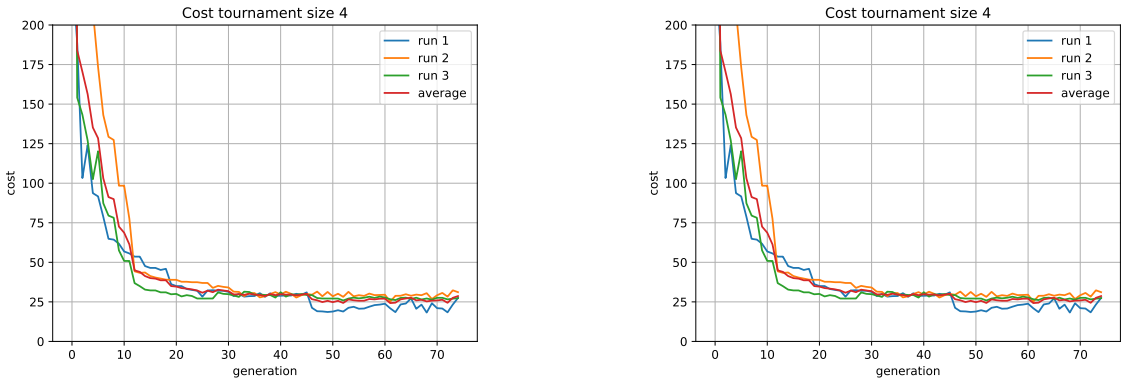


Figure 13.13: Cost and resemblance of 3 runs with tournament size 4

Here the code to calculate the diversity is shown.

```
class PotentialTournament:
def __init__(self):
    self.checker = CheckTree()  # Initialise the class that calculates the diversity between trees

def run(self, trees):
    samples = 8  # Amount of trees every tree is compared with
    pot_points = 0  # Set potential points to
    for i in range(len(trees) - 1 - samples):  # Compare every tree in the population
        for j in range(samples):  # Amount of trees to compare with
            if min([len(trees[i][0].nodes),
                    len(trees[i + j + 1][0].nodes)]) > 0:  # If both trees contain at least 1 node compare the trees
                pot_points +=
                    self.checker.calc_potential(trees[i][0].nodes, trees[i + j + 1][0].nodes) / min(
                    [len(trees[i][0].nodes), len(trees[i + j + 1][
                                                    0].nodes)])  # Calculate diversity between two trees and divide
                # by the amount of nodes in the smallest tree
            if min([len(trees[i][1].nodes), len(trees[i + j + 1][1].nodes)]) > 0:
                pot_points +=
                    self.checker.calc_potential(trees[i][1].nodes, trees[i + j + 1][1].nodes) / min(
                    [len(trees[i][1].nodes), len(trees[i + j + 1][1].nodes)])
            if min([len(trees[i][2].nodes), len(trees[i + j + 1][2].nodes)]) > 0:
                pot_points +=
                    self.checker.calc_potential(trees[i][2].nodes, trees[i + +j + 1][2].nodes) / min(
                    [len(trees[i][2].nodes), len(trees[i + j + 1][2].nodes)])
    pot_points = pot_points / (3 * samples) / (
                len(trees) - 1 - samples)  # Divide by amount of trees and amount of comparisons per tree
    return pot_points


class CheckTree:
def __init__(self):
    self.points_node = 1  # Points received for a correct node in subtree
    self.points_circuit = 1  # Points received for a correct circuit in subtree
    self.points_circuit2 = 1  # Points received for a correct circuit
    self.points_depth2 = 3  # Points subtracted for duplicate subtree of depth 2
    self.points_depth3 = 6  # Points subtracted for duplicate subtree of depth 3
    self.list1 = []
```

```
35      self.list2 = []
36      self.match = []
37      self.count = 0
38
39  """Calculate the diversity between two trees"""
40
41  def calc_potential(self, list1, list2):
42      self.match = []
43      points = 0
44      for i in list1:  # For every node check if the tree coming from that
            node is equal
45          for j in list2:
46              if i.type == j.type:
47                  t = self.check_tree(i, j)  # Create a list of matching
                        nodes
48                  if t:  # Check if the return is not False
49                      points += t[0]  # Add points
50                      self.match.append(i)  # Place the node in the match
                            list
51      points += self.check_circuits(list1, list2)  # Check if individual
            circuits match
52      points += self.duplicate()  # Subtract points for duplicates
53      return points
54
55  """Subtract points for duplicates"""
56
57  def duplicate(self):  # If subtrees of depth 2 are equal in the current
        tree, subtract 3 points
58      dup_penalty = 0
59      for i in range(len(self.match)):
60          for j in range(len(self.match) - i - 1):
61              if self.match[i] != 'False' and self.match[j + i + 1] !=
                    'False':  # Continue if the node is not already
62                  # marked as duplicate
63                  if self.check_equal2(self.match[i], self.match[j + i +
                        1]):  # Check if a tree of depth 3 is equal
64                      dup_penalty -= self.points_depth3  # If so remove
                            points equal to worth of depth 3 tree
65                      self.match[j + i + 1] = 'False'  # Mark the duplicate
                            as removed
66                  elif self.check_equal1(self.match[i], self.match[j + i +
                        1]):  # Check if a tree of depth 2 is equal
67                      dup_penalty -= self.points_depth2  # If so remove
                            points equal to worth of depth 3 tree
68                      self.match[j + i + 1] = 'False'  # Mark the duplicate
                            as removed
69      return dup_penalty
70
71  """Check if individual circuits match"""
72
73  def check_circuits(self, list1, list2):
74      points = 0
75      list1_2 = []
76      list2_2 = []
77      for i in list1:  # Add all circuits in tree1 to the list
78          if isinstance(i.branch[0], GenotypeCircuit):
```

```
79              list1_2.append(i.branch[0].type)
80          if isinstance(i.branch[1], GenotypeCircuit):
81              list1_2.append(i.branch[1].type)
82      for i in list2:  # Add all circuits in tree2 to the list
83          if isinstance(i.branch[0], GenotypeCircuit):
84              list2_2.append(i.branch[0].type)
85          if isinstance(i.branch[1], GenotypeCircuit):
86              list2_2.append(i.branch[1].type)
87      for i in range(len(list1_2)):
88          for j in range(len(list2_2)):
89              if list1_2[i] and list2_2[j]:  # Continue if the circuit is
                    not yet matched
90                  if list1_2[i] == list2_2[j]:  # Continue if both circuits
                        are equal
91                      points += self.points_circuit2  # Increase points
92                      list1_2[i] = False  # Remove circuit from list
93                      list2_2[j] = False  # Remove circuit from list
94                      break
95      return points
96
97  """Check if trees rooting from node1 and node2 have a matching depth2,
        depth3 or depth4 tree"""
98
99  def check_tree(self, node1, node2):
100     order = [[], [], []]
101     self.count = self.points_node  # Set the count to 1, because the node
            matches
102     """Depth 1, check if depth 2 is equal"""
103     r = self.check_equal(node1, node2)  # Check if the branches are equal
104     if r == 'False':  # If not return false
105         return False
106     else:
107         order[0] = r  # Define whether the left and right branches are
                swapped: '1' or not '0'
108     count2 = self.count  # Save the count for if the branches further
            down don't match
109     """Depth 2, check if depth 3 is equal"""
110     for i in range(2):
111         if isinstance(node1.branch[i], GenotypeNode):  # Only check if
                the next element is a node, not when it's
112             # a circuit
113             r = self.check_equal(node1.branch[i], node2.branch[i ^
                    order[0]])
114             if r == 'False':  # If no match return points for a subtree
                    of depth 2
115                 return [count2, node1]
116             else:
117                 order[1].append(r)  # Save whether branches are swapped
                        or not
118         else:
119             order[1].append('end')  # End means the function does not
                    need to check for matches below this node
120     count2 = self.count  # Save the count for if the branches further
            down don't match
121     """Depth 3, check if depth 4 is equal"""
122     for i in range(2):
```

```
123         for j in range(2):  # Depth 2 check if depth 3 is equal
124             if order[1][i] != 'end' and isinstance(order[1][i],
                    GenotypeNode):
125                 r = self.check_equal(node1.branch[i].branch[j],
126                                      node2.branch[i ^ order[0]].branch[j
                                             ^ order[1][j]])
127                 if r == 'False':
128                     return [count2, node1]
129                 else:
130                     order[2].append(r)
131             else:
132                 order[2].append('end')
133         count2 = self.count
134         return [count2, self.count]
135
136 """Check if the branches of the node are equal"""
137
138 def check_equal(self, node1, node2):
139     if node1.branch[0].type == node2.branch[0].type and
            node1.branch[1].type == node2.branch[1].type:  # Check if
140         # branches are equal
141         for it in range(2):  # Add points for node or circuit type
                depending on the component
142             if isinstance(node1.branch[it], GenotypeNode):
143                 self.count += self.points_node
144             else:
145                 self.count += self.points_circuit
146         return 0  # Branches are equal
147     elif node1.branch[0].type == node2.branch[1].type and
            node1.branch[1].type == node2.branch[0].type:  # Check if
148         # branches are equal but swapped
149         for it in range(2):
150             if isinstance(node1.branch[it], GenotypeNode):
151                 self.count += self.points_node
152             else:
153                 self.count += self.points_circuit
154         return 1  # Branches are swapped
155     else:
156         return 'False'
157
158 """Check if a depth 2 tree is equal but don't return information about
        swapped or non-swapped matches"""
159
160 def check_equal1(self, node1, node2):  # Check if the branches of the
        node are equal
161     if node1.branch[0].type == node2.branch[0].type and
            node1.branch[1].type == node2.branch[1].type:
162         for it in range(2):
163             if isinstance(node1.branch[it], GenotypeNode):
164                 self.count += self.points_node
165             else:
166                 self.count += self.points_circuit
167         return True  # Branches are equal
168     elif node1.branch[0].type == node2.branch[1].type and
            node1.branch[1].type == node2.branch[0].type:
169         for it in range(2):
```

```
170             if isinstance(node1.branch[it], GenotypeNode):
171                 self.count += self.points_node
172             else:
173                 self.count += self.points_circuit
174         return True  # Branches are swapped
175     else:
176         return 'False'
177 
178 """Check if a depth 3 tree is equal but don't count points"""
179 
180 def check_equal2(self, node1, node2):
181     order = [[], 0]
182     """Depth 1, check if depth 2 is equal"""
183     r = self.check_equal(node1, node2)  # Check if the branches are equal
184     if r == 'False':  # If not return false
185         return False
186     else:
187         order[0] = r  # Define whether the branches are swapped: '1' or
                not '0'
188     """Depth 2, check if depth 3 is equal"""
189     for i in range(2):
190         if isinstance(node1.branch[i],
191                       GenotypeNode):  # Only check if the next element is
                          a node, not when it's a circuit
192             r = self.check_equal(node1.branch[i], node2.branch[i ^
                    order[0]])
193             if r == 'False':
194                 return False
195             else:
196                 order[1] += 1
197     if order[1] > 1:
198         return True
199     else:
200         return False
```

13.5. Evaluator Class

Here the code of the evaluator is shown. This class takes as input the costs of the population and outputs which individuals to select.

```
1  class Evaluation:
2      def __init__(self, pop_size: int, sel_size: int):
3          self.pop_size = pop_size    # Population size
4          self.sel_size = sel_size    # Amount of selected parents
5          self.tournament = TournamentSelection(self.pop_size,
               self.sel_size)
6  
7      def evaluate(self, costs: list[float], tournament_size, i) ->
           list[int]:
8          c: np.ndarray = np.array(list(enumerate(costs)))
9          return self.tournament.select(c, i, tournament_size)
10 
11 
12 class TournamentSelection:
13     # Input of costs should be an enumerated list, so n x 2 matrix where
          one is the number of
```

```
    # the circuit and two the cost

    def __init__(self, pop_size: int, sel_size: int, set_size: int = 3):
        self.sel_size = sel_size  # Amount of selected parents
        self.set_size = set_size  # Amount of individuals in tournament set
        self.pop_size = pop_size  # Population size

    def select(self, costs: np.ndarray, i, tournament_size) -> list[int]:
        # Costs is an array of enumerated costs of the incoming circuits.
        selected = [0 for _ in range(self.sel_size)]    # Create a list where the selected parents are placed later
        if i > 300:
            tournament_size = 5

        def tournament():   # Tournament selection function
            for i in range(self.sel_size):
                sel = np.random.choice(self.pop_size, tournament_size, replace=False)   # Randomly select tournament_size amount of individuals from the population
                tournament = costs[sel, :]  # Set the costs of the selected individuals in the tournament
                sel = np.argmin(tournament[:, 1])   # Select the individual with the smallest cost
                selected[i] = int(tournament[sel, 0])   # Place the number of the winning individual in the selected list
            return selected

        selected = tournament()

        return selected
```

13.6. Figures Noise

Tournament size 3, 5 and 10 with noise have all been run 3 times. The results of the added noise to tournament selection can be found in figure 13.14, 13.15 and 13.16.

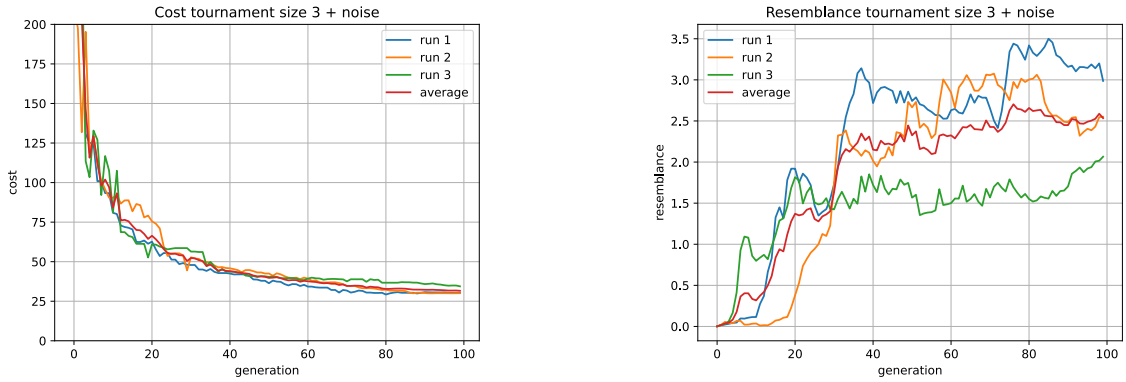


Figure 13.14: Cost and resemblance of 3 runs with tournament size 3 + noise

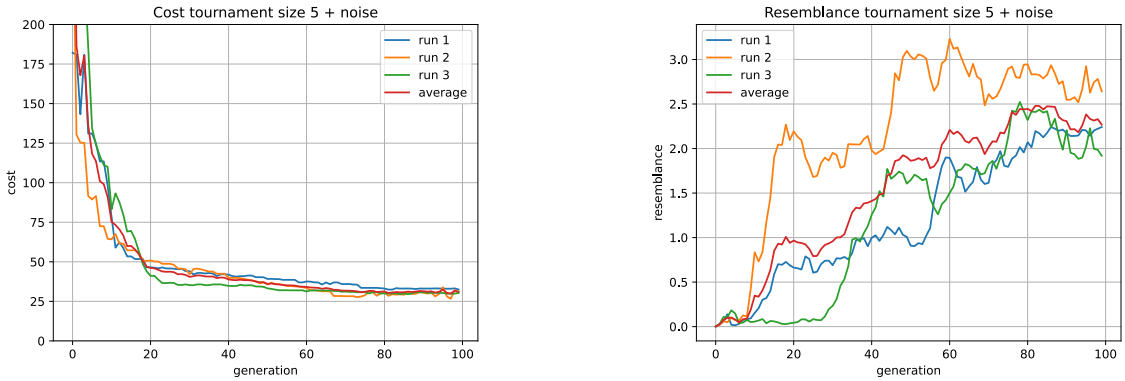


Figure 13.15: Cost and resemblance of 3 runs with tournament size 5 + noise

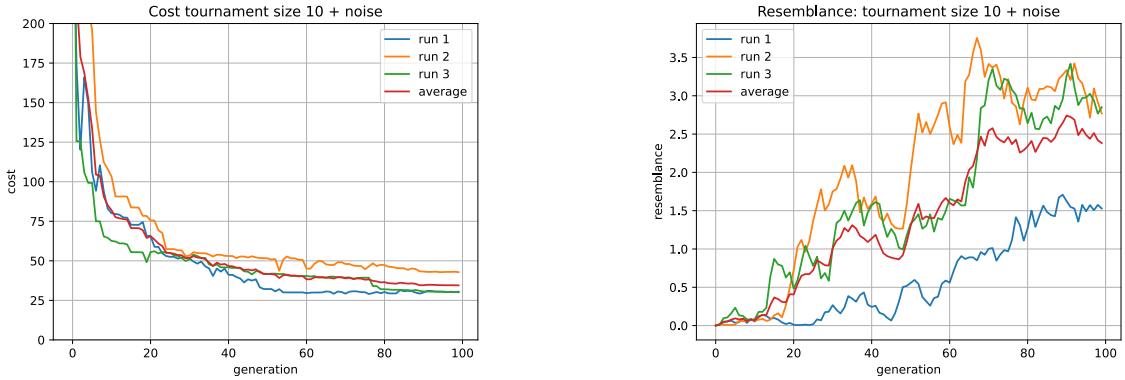


Figure 13.16: Cost and resemblance of 3 runs with tournament size 10 + noise

Bibliography

[1] O. Aaserud and I. Ring Nielsen. “Trends in Current Analog Design—a Panel Debate”. In: *Analog Integr. Circuits Signal Process.* 7.1 (Jan. 1995), pp. 5–9. ISSN: 0925-1030. DOI: 10.1007/BF01256442. URL: https://doi.org/10.1007/BF01256442 (cit. on p. 4).

[2] De Jong Kenneth Alan. *An analysis of the behavior of a class of genetic adaptive systems*. University Microfilms International, 1975 (cit. on p. 23).

[3] T. Back. “Selective pressure in evolutionary algorithms: a characterization of selection mechanisms”. In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. 1994, 57–62 vol.1. DOI: 10.1109/ICEC.1994.350042 (cit. on pp. 11, 24).

[4] Hoffmeister.F Bäck.T. “Extended selection mechanisms in genetic algorithms”. In Proceedings of the Fourth International Conference on Genetic Algorithms 1991; 92-99.” In: () (cit. on p. 11).

[5] James Edward Baker, ed. *Proceedings of an International Conference on Genetic Algorithms and their applications*. Psychology Press, 1985, Adaptive selection methods for genetic algorithms (cit. on p. 14).

[6] Tobias Blickle and Lothar Thiele. “A comparison of selection schemes used in evolutionary algorithms”. In: *Evolutionary Computation* 4.4 (1996), pp. 361–394. DOI: 10.1162/evco.1996.4.4.361 (cit. on pp. 11, 13–16).

[7] Federico Castejón and Enrique J Carmona. “Automatic design of analog electronic circuits using grammatical evolution”. In: *Applied Soft Computing* 62 (2018), pp. 1003–1018 (cit. on pp. 4, 5).

[8] Wai-Kai Chen. *Passive, Active, and Digital Filters, third Edition*. 2nd ed. The Circuits and Filters Handbook, 3rd Edition. Boca Raton, FL: CRC Press, June 2009 (cit. on p. 4).

[9] P. Coelho et al. “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 (cit. on pp. 4, 5).

[10] Kalyanmoy Deb et al. “Multi-objective evolutionary algorithm for land-use management problem”. In: *International Journal of Computational Intelligence Research* 3.4 (2007). DOI: 10.5019/j.ijcir.2007.118 (cit. on p. 25).

[11] Vance Dickason and E A J Bogers. *Luidsprekerkasten ontwerpen*. Segment B.V., 1996 (cit. on p. 3).

[12] Yongsheng Fang and Jun Li. “A Review of Tournament Selection in Genetic Programming”. In: *Advances in Computation and Intelligence*. Springer Berlin Heidelberg, 2010, pp. 181–192. DOI: 10.1007/978-3-642-16493-4_19. URL: https://doi.org/10.1007/978-3-642-16493-4_19 (cit. on p. 15).

[13] 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 (cit. on pp. 4, 5).

[14] Stanley Gotshall and Bart Rylander. In: *Optimal Population Size and the Genetic Algorithm* () (cit. on p. 23).

[15] John J. Grefenstette and James E. Baker. “How Genetic Algorithms Work: A Critical Look at Implicit Parallelism”. In: *International Conference on Genetic Algorithms*. 1989 (cit. on p. 14).

[16] Xiaosong Han et al. “An Efficient Genetic Algorithm for Optimization Problems with Time-Consuming Fitness Evaluation”. In: *International Journal of Computational Methods* 12.01 (Jan. 2015), p. 1350106. DOI: 10.1142/s0219876213501065 (cit. on pp. 4, 5).

[17] Jingsong He, Pengfei Xia, and Wen 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 (cit. on pp. 4, 5).

[18] Jingsong He and Jin 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 (cit. on pp. 4, 5).

[19] Na He, Dianguo Xu, and Lina Huang. “The Application of Particle Swarm Optimization to Passive and Hybrid Active Power Filter Design”. In: *IEEE Transactions on Industrial Electronics* 56.8 (2009), pp. 2841–2851. DOI: 10.1109/TIE.2009.2020739 (cit. on p. 4).

[20] John H Holland. *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press, Dec. 1975 (cit. on pp. 5, 14).

[21] Khalid Jebari and Mohammed Madiafi. “Selection methods for genetic algorithms”. In: *International Journal of Emerging Sciences* 3.4 (2013), pp. 333–344 (cit. on pp. 11, 14).

[22] Min Jiang, Zhenkun Yang, and Zhaohui 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 (cit. on p. 4).

[23] Hao Lan and Jingsong He. “Increasing-Dimension Evolution: Make the evolutionary design of passive filters more efficient”. In: *Applied Soft Computing* 131 (2022), p. 109740. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2022.109740 (cit. on p. 4).

[24] K.-H. Lee et al. “A novel approach in parameter adaptation and Diversity Maintenance for genetic algorithms”. In: *Soft Computing - A Fusion of Foundations, Methodologies and Applications* 7.8 (2003), pp. 506–515. DOI: 10.1007/s00500-002-0235-1 (cit. on p. 16).

[25] Brian McGinley et al. “Maintaining healthy population diversity using adaptive crossover, mutation, and selection”. In: *IEEE Transactions on Evolutionary Computation* 15.5 (2011), pp. 692–714. DOI: 10.1109/tevc.2010.2046173 (cit. on pp. 16, 37).

[26] Rayan Mina, Chadi Jabbour, and George E. Sakr. “A Review of Machine Learning Techniques in Analog Integrated Circuit Design Automation”. In: *Electronics* 11.3 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11030435 (cit. on p. 4).

[27] Hari Mohan Pandey, Ankit Chaudhary, and Deepti Mehrotra. “A comparative review of approaches to prevent premature convergence in GA”. In: *Applied Soft Computing* 24 (Nov. 2014), pp. 1047–1077. DOI: 10.1016/j.asoc.2014.08.025. URL: https://doi.org/10.1016/j.asoc.2014.08.025 (cit. on p. 11).

[28] D T Pham and M Castellani. “Adaptive Selection Routine for Evolutionary Algorithms”. In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 224.6 (Sept. 2010), pp. 623–633. DOI: 10.1243/09596518jsce942. URL: https://doi.org/10.1243/09596518jsce942 (cit. on pp. 11, 31).

[29] John Pillans. “Efficiency of evolutionary search for analog filter synthesis”. In: *Expert Systems with Applications* 168 (2021), p. 114267. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.114267. URL: https://www.sciencedirect.com/science/article/pii/S0957417420309787 (cit. on p. 4).

[30] Žiga Rojec, Jernej Olenšek, and Iztok Fajfar. “Analog Circuit Topology Representation for Automated Synthesis and Optimization”. In: *Electronic Components and Materials* 48.1 (2018), pp. 29–40 (cit. on p. 4).

[31] Simo Särkkä and Antti Huovilainen. “Accurate Discretization of Analog Audio Filters With Application to Parametric Equalizer Design”. In: *Audio, Speech, and Language Processing, IEEE Transactions on* 19 (Dec. 2011), pp. 2486–2493. DOI: 10.1109/TASL.2011.2144970 (cit. on p. 4).

[32] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. “Comparative review of selection techniques in genetic algorithm”. In: *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*. 2015, pp. 515–519. DOI: 10.1109/ABLAZE.2015.7154916 (cit. on p. 14).

[33] Y.R. Tsoy. “The influence of population size and search time limit on genetic algorithm”. In: *7th Korea-Russia International Symposium on Science and Technology, Proceedings KORUS 2003. (IEEE Cat. No.03EX737)*. Vol. 3. 2003, 181–187 vol.3 (cit. on p. 23).

[34] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697. DOI: 10.5555/1593511 (cit. on p. 5).

[35] Revna Acar Vural and Tulay 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 (cit. on p. 4).

[36] Huayang Xie and Mengjie Zhang. “Sampling Issues of Tournament Selection in Genetic Programming”. In: (Dec. 2013) (cit. on pp. 15, 26).

[37] Xuesong Yan et al. “Electronic Circuit Automatic Design Based on Genetic Algorithms”. In: *Procedia Engineering* 15 (2011), pp. 2948–2954. DOI: 10.1016/j.proeng.2011.08.555 (cit. on pp. 4, 5).

[38] Hank Zumbahlen. *Linear Circuit Design Handbook*. London, England: Newnes, Feb. 2008 (cit. on p. 4).