Original scientific paper

https://doi.org/10.33180/InfMIDEM2019.405

Informacije MIDEM
Journal of Microelectronics,
Electronic Components and Materials
Vol. 49, No. 4(2019), 229 – 239

Grammatical Evolution-based Analog Circuit Synthesis

Matevž Kunaver

University of Ljubljana, Faculty of Electrical Engineering, Ljubljana, Slovenia

Abstract: Computer aided circuit design is becoming one of the mainstream methods for helping circuit designers. Multiple new methods have been developed in this field including Evolutionary Electronics. A lot of work has been done in this field but there is still a room for improvement since some of the solutions lack the flexibility (diversity of components, limited topology etc.) in circuit design or lack complex fitness functions that would enable the synthesis of more complex circuits. The research presented in this article aims to improve this by introducing Grammatical Evolution-based approach for circuit synthesis. Grammatical Evolution offers great flexibility since it is rule based – adding a new element is as simple as writing one additional line of initialization code. In addition, the use of a complex multi-criteria function allows us to create circuits that can be as complex as required thus further increasing the flexibility of the approach. To achieve this, we use a combination of Python and SPICE to create a series of netlists, evaluate them in the PyOpus environment, and select the best possible circuit for the task. We demonstrate the efficiency of our approach in three different case studies where we automatically generate oscillators and high/low-pass filters of second and third order.

Keywords: Automated synthesis; analog circuits; grammatical evolution; computer-aided design; evolutionary algorithms

Sinteza analognih vezij s pomočjo slovnične evolucije

Izvleček: Računalniško podprto načrtovanje vezij postaja eno ključnih orodij načrtovalcev elektronskih vezij. Na tem področju se je v zadnjem času pojavilo mnogo novih pristopov, kot na primer evolucijska elektronika. Kljub temu, da se področje živahno razvija, pa so možne in tudi potrebne še mnoge izboljšave. Zlasti zato, ker marsikatera obstoječa tehnika ne nudi dovolj prilagodljivosti pri sintezi vezij (omejen nabor elementov, omejitve pri topologijah ipd.) ali pa ne nudi možnosti za razvoj bolj kompleksnih vezij. S pristopom, ki uporablja tako imenovano slovnično evolucijo (angl. grammatical evolution), želimo te pomanjkljivosti odpraviti. Slovnična evolucija je izjemno prilagodljiva tehnika, ki deluje po principu pravil (t.j. ukazov, s pomočjo katerih se izgradi posamezen element vezja). Zato dodajanje novega tipa elementa v sistemu, ki uporablja slovnično evolucijo, ni nič bolj zapleteno kot vnos dodatne vrstice v inicializacijsko kodo. Poleg tega smo pri našem pristopu uporabili večkriterijsko funkcijo, ki nam omogoča sintezo poljubno kompleksnega vezja. Celoten sistem smo razvili in preizkusili v programskih okoljih Python in SPICE, s pomočjo katerih smo ustvarili serijo datotek z opisi vezij (angl. netlist), jih ovrednotili v okolju PyOpus ter s pomočjo kriterijske funkcije izbrali vezje, ki je najboljše za zadano nalogo. Uporabnost naše metode smo prikazali na treh primerih, kjer smo avtomatsko sintetizirali oscilatorje ter visoko in nizko prepustna sita drugega in tretjega reda.

Ključne besede: Avtomatska sinteza, analogna vezja, slovnična evolucija, računalniško podprto načrtovanje, evolucijski algoritmi

** Corresponding Author's e-mail: matevz.kunaver@fe.uni-lj.si*

1 Introduction

Analog circuit design has gradually changed from manual design (in the 1970s) to computer assisted design where a highly skilled engineer uses an assortment of computer tools to create a circuit with the specified characteristics. These tools range from simple schematic design to accurate circuit simulators (Simulation Program with Integrated Circuit Emphasis – SPICE [1]) that can simulate circuit behaviors and thus show if a circuit is actually worth implementing. This of course greatly reduces both the material costs (since one must only implement the final, best circuit) and the time to production since the simulations require only a fraction of time compared to actually creating and measuring each potential circuit.

These simulations, however, still require a lot of expertise from the user since he/she must still be able to manually specify things like the desired topology and necessary elements [2, 3, 4]. The rise of easily accessible (and more powerful) computers led to the research and development of tools that are capable (to some degree) of creating the desired topology automatically given a list of possible elements (resistors, coils, generators etc.) [5, 6] and circuit characteristics. Such tools are of great interest since they streamline the design process, save a lot of time and also grant the possibility of circuit design to a user who might not be well-versed in physical circuit design.

These tools then led to the development of Electronic Design Automation (EDA) [7] and Evolutionary Electronics [8], which allow automatic circuit synthesis and optimization. The idea behind this approach is to have the engineer simply specify the characteristics in an appropriate format (a cost function) and have the system create the appropriate circuit without any further interaction. This was first made possible when Koza [9] created topologies in 1992 with arbitrary connections using genetic programming (GP). Genetic programming allows a great diversity of developed topologies and offers great flexibility when selecting a cost function and the topology components. The approach quickly spread to other research groups [10, 11, 12], who tackled issues such as bloat (an excessive growth of circuits with surplus elements, such as two or more elements of the same type in series or parallel) and alternative topology representations [12, 13].

One of such topology representations is in the form of formal grammar to be used by Grammatical Evolution (GE), an evolutionary computation technique related to the idea of GP. GE offers great flexibility and simplicity during the initialization of the problem. It was already successfully used for simple circuit generation by Castejon et al. [6] but lacked a more complex fitness function. In this paper, we propose a combination of the GE approach together with a complex fitness function, similar to the one proposed by Rojec et al. [5]. The use of such function allows fine-tuning of several parameters at the same time (slope, gain, cut-off frequency etc.) but was so far limited only to a matrix-based GP approach. We believe that the combination of GE and an appropriate cost function should result in a tool that would allow a simple and efficient generation of complex circuits that meet several different criteria at once.

The structure of the paper is as follows. Section 2 covers the methods used for our circuit representation – SPICE syntax, fitness functions, GE production rules and population manipulation techniques. Section 3 presents the results of three different case studies that we performed during the development process, and section 4 summarizes our findings and compares them to three other techniques.

2 Materials and methods

Our goal is to create a system that can synthesize a circuit given two input parameters: (i) a list of acceptable components (resistors, capacitors etc.) and (ii) the desired circuit characteristic interpreted as a GE fitness function.

The final system should not require any in-depth knowledge of circuit design beyond being capable of formulating the desired circuit response and finding the circuit models in the SPICE environment.

We rely on a combination of Python (in which we implemented the GE algorithm), SPICE (for circuit evaluation) and PyOpus [14] as the link between them. During initialization we:

- Specify the components that can be used in the circuit (resistor, capacitor, source…) using the SPICE format.
- Define the fitness (RMSE, multi-criteria, PyOpus measurements etc.).
- Define the GE grammar production rules.
- Set the GE parameters:
  - Number of generations – how many groups of programs to evaluate.
  - Population size – how many individuals (sub-circuits) are produced in each generation.
  - Crossover rules – which nodes can be replaced
  - Mutation probability
  - Elite size – what percentage of best individuals make it into the next generation.

Once all the parameters are set, we run the experiment by performing the following steps:

1. Evolve a sub-circuit for each genome sequence in the current generation.
2. Create a netlist for the generated circuit.
3. Evaluate the sub-circuit using the selected fitness function.
4. Trim the population – keep the best 10%.
5. Create a new generation of circuits by combining the previous best 10% and generating the rest with mutation, crossover and selection (see 2.2.4).
6. Repeat this until we have created and evaluated the desired number of generations.

2.1 Circuit representation

Since we use SPICE to simulate and evaluate our circuits, we use the SPICE notation (netlists) for our circuits to make the procedure as simple as possible. Our circuit consists of a "main" circuit that features the necessary evaluation elements (sources, loads, ground ports etc.) and the "evaluation" subcircuit, which was generated by our algorithm. The "main" circuit is shown in Figure 1.

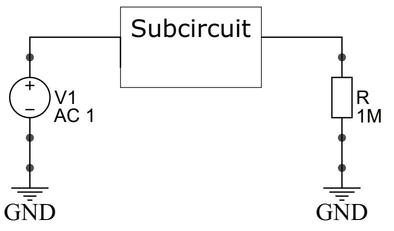

Figure 1: Main circuit for evaluation of the generated subcircuits.

The two circuits are stored in separate netlist files and used as input parameters for our PyOpus simulation.

2.2 Grammatical evolution

Grammatical Evolution is one of the emerging methods from the field of Evolutionary computation [6, 15, 16]. The approach is based on using a grammar that consists of production rules for each possible circuit element and its characteristics. The definition of the grammar structure is one of the most important steps when using the GE approach. Each circuit (i.e. filter, oscillator, amplifier circuit) requires a different grammar structure since it can contain different elements, ports, and so on. The grammar is usually defined using the Backus-Naur form (BNF).

Once we select the grammar, we must also select the evolution hyper-parameters such as the population size, crossover type and mutation probability. These parameters impact the duration of the simulation (a larger population requires more time for evaluation) and the success rate of each run (larger population and more frequent mutations can cover more of the search-space and possibly find a better solution).

All our case studies featured a population size of 300 individuals per generation, 250 generations per run and a fixed mutation rate of 5%.

2.2.1 The Grammar and production rules

The grammar used for this article was designed to accommodate future expansions and modifications, i.e. to be as flexible as possible when adding new components. An interesting thing to note is that although we created these rules from scratch we ended up with rules that were quite similar to those used by Castejon et al. [6].

The grammar consists of rules formulated in BNF format. These rules either generate non-terminal nodes (components) or terminal node that represent component characteristics such as component type (resistor, coil and capacitor), values (i.e. resistance, capacity) and ports. The names of elements are generated later (see 2.3 for more details). The common rules used in all our case studies are listed in Table 1.

Table 1: Genetic grammar production rules.

Rule	Possible values
<part>	"<cap>"|"<res>"|"<coi>"|" "
<res>	"rXX (<gPair>) <num>e<n>"
<cap>	"cXX (<gPair>) <num>e-<n>"
<coi>	"lXX (<gPair>) <num>e-<n>"
<gPair>	"input 0"|"input 1"|"input 2"|"input 3"|"input 4"| "input output"|"1 2"| "1 3"|"1 4"|"1 0"|"1 output"|"2 3"| "2 4"|"2 0"|"2 output"|"3 4"|"3 0"| "3 output"|"4 0"|"4 output"
<num>	"<n>"|"<n><n>"
<n>	"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"| "9"|"0"

The starting symbol (see Table 2) is then used to generate the number of elements present in the subcircuit. This can be done recursively where one generates nodes until a terminal node is reached or one achieves the maximum possible depth. Alternatively, we can use an iterative approach where we set a maximum number of components (as opposed to maximum depth in the recursive approach). Here we deviated from the grammar form used in [6] as we used a different approach to keeping the number of elements within a preset maximum number. Castejon uses a dynamic option where a codon can either add an additional element or not. In our approach, we set the maximum number of elements and let the codons select whether or not an element exists. This was done to limit the circuit bloat.

2.2.2 Individuals and chromosomes

Grammatical Evolution creates individuals using a sequence of chromosomes. Each chromosome sequence contains 300 randomly generated chromosomes (a random integer between 1 and 256). These chromosomes are then interpreted using the GE rules (see 2.2.3 for an example). The same sequence of chromosomes will always generate the same subcircuit as long as the production rules remain the same. This greatly simplifies storage and reproduction of our results since we do not need to store the actual netlists, files or objects but simply need to store simple sequences of 300 integers thus greatly reducing the required storage space.

2.2.3 Demo Sequence

An example of individual generation would therefore proceed as follows. The system first creates a chromosome sequence {229, 52, 125, 40, 60, 99, 100….} and uses a starting sequence as shown below.

"<part><part><part>"

The algorithm then focuses on replacing the first symbol in the sequence. The symbol "<part>" has three possible values which is why the algorithm then uses modulo operation on the current genome (229%3) which gives the result 1 which is the second of the possible values. The "<part>" symbol is therefore replaced with the "<res>" symbol.

"<res><part><part>"

The resistor has only one possible value so the algorithm proceeds with a modulo 1 operation (52%1) which returns 0 and thus selects the only possible resistor type.

"rXX (<gPair>)<num>e<n><part><part>"

The next symbol in the sequence is then the "<gPair>" which has 20 possible values. Using modulo 20 on the next genome in the sequence (125%20) returns 5 which means that the resistor is set to be connected between the input and output ports of the subcircuit.

"rXX (input output)<num>e<n><part><part>"

The next genome (40%2) sets the "<num>" part to a single digit of "<n>".

"rXX (input output)<n>e<n><part><part>"

Then the next genome selects one of the ten possible values for the digit (60%10) and sets it to "1".

"rXX (input output) 1e<n><part><part>"

Lastly, the value of exponent is set to "0" using the next genome and selecting between the 10 possible values (99%10).

"rXX (input output)1e0<part><part>"

The first element is therefore a resistor with resistivity of one 1 Ohm and connected between the input and output port.

Having set all the parameters of the first "<part>" symbol, the algorithm then returns to the start symbol and uses the next genome (100%3) to select the type of the next element, which would in this case again be a resistor. This continues until all the symbols have been replaced with their parameter values.

2.2.4 Population manipulation

Once all the individuals in the current population are evaluated and sorted, the question of producing the next generation occurs. This is done using several manipulation techniques. Using experience and advice from other experiments [17, 5, 18] we take the best individuals from the current generation and move them into the next until we fill one tenth of it (so in our case of 300 individuals per generation we allow the best 30 individuals to proceed into the next one). Since we already evaluated these individuals, we will not need to do so again.

Next, we check if any of the (non-elite) individuals will mutate (a 5% chance in our case). When mutating, the algorithm select one random node of the individual and remove any nodes connected to it. The chromosome corresponding to this node will then be randomly changed to a new value. Afterwards the GE rules will be used to re-create the mutated individual. A mutation can therefore result in a completely new circuit or a minor change in the circuit such as changing the numeric value of the element or the ports to which it is connected. So for example, if we begin with the following sequence:

"rXX (input output)1e10 cXX(1 2) 4e-3"

Mutation can either change one of the parameters of the elements (for example the capacitivity of the capacitor)

"rXX (input output)1e10 cXX(1 2) 10e-6"

Or completely replace one the elements with a new one (for example replace the resistor with a new capacitor)

"cXX (1 output)2e-9 cXX(1 2) 4e-3"

Following the mutation sequence, the algorithm will perform a crossover function on all the remaining individuals. This is done by selecting two individuals and randomly selecting a node in the first one. We then check if we can find a node of the same type in the second individual. If we do, we switch them between the two individuals. If not, we leave the individuals as they are. The level of exchange can be set to high level elements only (exchange complete elements with all attributes) or any level desired (exchange ports, numeric values etc.). The resulting individuals replace the originals. An example of crossover can be shown using the following two sequences which represent two individuals:

"cXX (1 output) 2e-9 cXX (1 2) 4e-3"

"rXX (input output) 1e10 cXX (1 2) 10e-6"

Our algorithm would then decide to crossover on the "<cap>" node, meaning that it would try to find a node of this type in each individual. Let's assume that it selects the first capacitor in the first individual and the only capacitor in the second individual. The algorithm then swaps the two and stores them as the new individuals resulting in:

"cXX (1 2) 10e-6 cXX (1 2) 4e-3"

"rXX (input output) 1e10 cXX (1 output) 2e-9"

Lastly, after selection we check if the new generation contains enough diversity. Without doing this, we would quickly find that most of the individuals contain the same circuit with only minor differences (for example, a resistor of 10 Ohms instead of 9 Ohms). While this could be useful when optimizing the final solution, it can quickly lead us into a dead-end of the search space (a local minimum of the fitness function). We therefore check the diversity of population every 5 generations and remove any duplicate individuals that we find. All such duplicates are then replaced with a "fresh" randomly generated circuits which will (hopefully) increase the diversity and thus the chance of finding the best possible solution. The frequency of this can be as high or as low as we require but we found that when working with simple circuits, it is beneficial to do this as frequently as possible.

Once all these steps are done, the next generation is complete and ready for evaluation.

2.3 Netlists and PyOPUS

Once an individual is transformed into a string sequence using the production rules (for example "cXX (1 output) 3e-8 cXX (1 2) 39e-5 cXX (input 2) 73e-2 cXX (3 output) 2e-7 rXX (1 0) 8e3 rXX (2 0) 4e7 rXX (1 3) 07e8 rXX (3 0) 96e2 rXX (input output) 06e9 cXX (2 0) 07e-7") we need to transform this sequence into a suitable subcircuit for the PyOpus simulator. Only then are we able to evaluate it (calculate its cost function). To do this, we utilize a simple string parser that performs two important tasks:

- Create a unique name for each of the circuit elements (change the first cXX to c01, the second to c02 and so on)
- Flag any circuit containing illegal nodes as faulty and not appropriate for simulation (for example cXX (2 2) 07e-9 is a capacitor that is connected to a single loop). Faulty circuits can otherwise loop the simulator, resulting in lost processing time.

If the circuit is faulty, the individual is not evaluated and has its cost set to the maximum possible value. Otherwise, the processed string is stored into a temporary file along with a header containing all the required SPICE subcircuit characteristics. This file is then used with the SPICE simulator during the evaluation procedure.

2.4 Fitness functions

The core of the GE approach is the fitness function used to evaluate individuals. This function can be as simple or as complex as desired but must provide a clear (numerical) fitness value of each individual in the generation. The better the value (usually meaning the lowest possible value) the better the individual and the better the chance for this individual to be the best possible solution for the problem at hand.

In our first case study, we used the same approach as Castejon et al [6] – a curve fitting metric. However, instead of using a custom weighted function, we used the "standard" form of curve comparison – RMSE. This approach has proven to be viable in the past [19] and meets the GE fitness function requirements – i.e. a smaller value is better. An example result when using RMSE as the fitness function is shown in Figure 2.

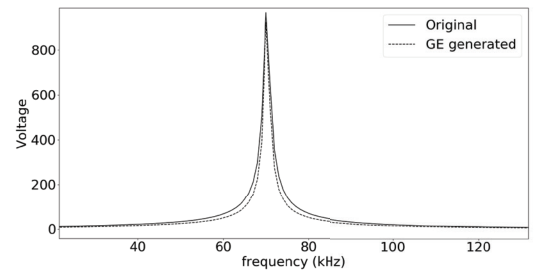

Figure 2: A comparison of performance between a voltage oscillator generated by our algorithm (dashed) and the original circuit (solid).

Once we moved to more complex circuits (filters of second and third order), we quickly discovered that simple fitness functions do not work sufficiently and lead to a low success rate. An example of a third-order filter design using RMSE is shown in Figure 3.

The problem stems from the fact that such a fitness function simply compares the difference between two curves on a point-by-point basis and is unable to in-

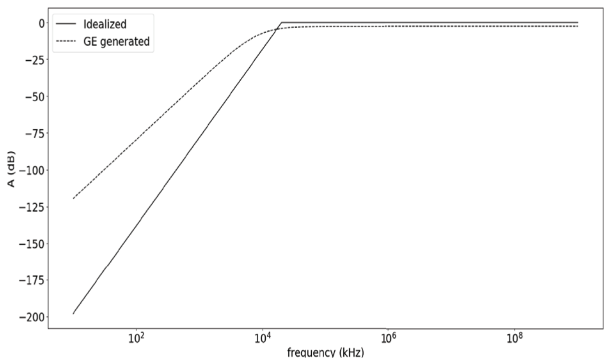

Figure 3: A failure when creating the third-order filter using an RMSE fitness function.

clude any additional design requirements as for example the desired level of dampening. Even using a weighted version of such a function (as in [6]) does not help.

We therefore designed a different (multi criteria) fitness function as suggested by Rojec et al. [5]. The proposed function allows focus on several characteristics at the same time and also allows assigning different priorities to each of them. Once we switched to the new fitness function, we were able to generate working filters which matched the desired characteristics quite nicely as seen in Figure 4.

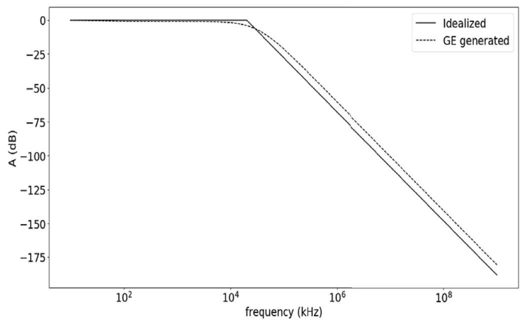

Figure 4: A successfully generated low-pass filter of the second order.

An additional advantage of the new fitness function was also the fact that we no longer required a comparison curve. When using RMSE we had to manually create a comparison curve, which meant that we had to have a comparison circuit ready. Using this baseline circuit we performed the PyOpus analysis and stored the results for further comparison. This is of course a bit controversial since it means that we had to have at least one example of a working circuit in order to be able to find other possible solutions. This can of course become a problem when dealing with more complex circuits or even when dealing with a user who does not have necessary knowledge.

Using the new fitness function we simply had to specify the desired characteristics (see 3.2.2) and the GE algorithm was able to run. As an additional bonus, we also sped up the evaluation procedure by performing our evaluations during the simulation itself. The speed increase was noticeable since we were now able to produce the final circuit in ten minutes or less (as compared to one hour reported by Rojec et al.).

3 Case studies

3.1 Oscillator circuit

In the first case study, we wanted to test several GE rule sets and see if they can produce feasible and workable circuits. We focused on replicating the performance of an oscillator circuit. We used RMSE as the fitness function and compared the voltage curve of the original oscillator with the GE generated curve. Figure 5 shows an example result where the solid line represents the original circuit, while the dashed one represents the GE generated circuit.

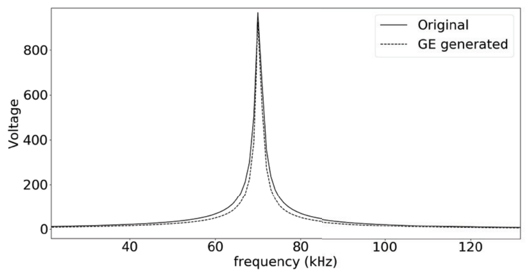

Figure 5: An RMSE generated oscillator voltage response.

In the evolution process, we used three different rule sets. The first set featured pre-set elements (one resistor, capacitor and coil – see rule (i) in Table 2) with the GE algorithm focusing on finding the correct element values. The idea behind this set was proving that our proposed technique actually finds a possible solution even when faced with severe limitations (in the form of a fixed circuit).

The second set allowed the algorithm to create as many components as possible (up to 20). We only limited the number of available ports and combinations by using the <gPair> element from Table 1. The start symbol (shown as (ii) in Table 2) therefore featured 20 <part> elements for which the GE algorithm chose whether or not they translated into an actual component.

The last rule set further relaxed the constraints and allowed any number of elements and any number of ports. The start symbol for this case became recursive as shown in Table 2. This means that, each time the GE algorithm created the next component, there was a 50% chance (since there are two possibilities in the <p> symbol) of creating an additional component and 50% chance of this being the last component in the circuit.

Table 2: Oscillator circuit grammar.

Rule	Possible values
(i) <start>	"r01 (input output) <num->e<n> c01 (input output) <num>e<n> l01 (input output) <num>e<n>"
(ii) <start>	"<-part-><-part->…<part>"
(iii) <start>	"<-p->"
(iii) <p>	"<part><p>"|"<part>"

Each of the rule sets resulted in a circuit that matched the original curve almost perfectly as seen in Figure 5. A sample circuit generated with the last set of rules is shown in Figure 6.

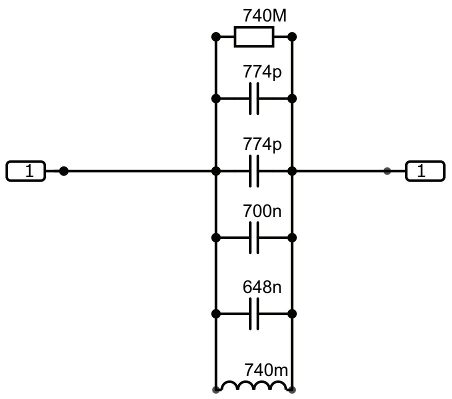

Figure 6: A GE generated oscillator circuit.

The results show that the GE algorithm is up to the task of creating the desired circuit. We can, however, see that there is a potential problem with bloat, since the third set of rules created a large number of components in most cases (an oscillator normally requires only three components). This can be alleviated during post-processing by analyzing the netlist and replacing parallel/serial elements with their equivalents.

3.2 Second-order filters

The first case study showed that we can generate simple circuits using our GE based approach. We then moved to a more complex example – second-order low/high-pass filters. As before, the aim remained the same – to automatically generate a filter with the desired characteristics. At the beginning, we retained the RMSE fitness function (and had to generate a comparison circuit for each example) but we quickly discovered that this fitness function did not seem to have a very high success rate – while we were always able to create a circuit with filter-like characteristics (i.e., with a cut-off frequency and dampening) we were unable to do so in a consistent manner. Upon reflection (see 2.2 for more detail) we switched to a multi-criteria fitness function as suggested by Rojec et al. [5].

At the beginning of the study, we focused on high-pass filters but later also generated low-pass versions to demonstrate the flexibility of our approach.

3.2.1 Using RMSE

We used a pre-set circuit to generate a comparison curve that was used to evaluate the GE generated circuits. We also used experience from the first case study to limited our rules to using up to 12 different circuit components to reduce bloat (limiting the upper number of components was also suggested in [5]). In addition, we removed the coil element from the component list, since a filter circuit usually consists of only resistors and capacitors.

We noticed after several extensive runs that there appeared to be some issues with the success rate of the algorithm. While it did find a possible solution in some of the runs, it quite often either completely failed or produced something that did not resemble a second-order filter at all, exhibiting quite a high cost function value. After analyzing several of the results, we came to the conclusion that the issue lies in the nature of the RMSE fitness function as discussed in 2.4.

We therefore switched to a more complex fitness function that allowed us to emphasize important aspects of the filter transfer functions and hopefully produce better (and more consistent) results.

3.2.2 Multi-criteria fitness function

We based our approach on the fitness function presented by Rojec et al. [5]. For the second-order filter, we focused on gain, cut-off frequency, ripple, and damping. Gain measures an increase (i.e., amplification) in the voltage level before the dampening begins. Gain of an ideal filter is equal to zero, meaning that the input level is stable before any changes applied by the filter. We calculated gain using this equation:

$$g = \left|0\,dB - gain\right| \tag{1}$$

The cut-off frequency indicated the frequency at which the damping begins. We set it to 20 kHz in our case and calculated the difference between this value and the frequency created by our algorithm using the following equation:

$$f_{off} = \left|\log_{10} 20\,kHz - \log_{10} f_{pass}\right| \quad (2)$$

Ripple indicates whether or not the input level before the cut-off frequency remains stable (i.e. the whole bandwidth is amplified at the same level). We calculated the ripple level using equation 2.

$$r = \begin{cases} ripple - 0.5\,dB, ripple > 0.5\,dB \\ 0, ripple \leq 0.5\,dB \end{cases} \quad (3)$$

Last but not least the damping indicates whether or not the designed filter actually achieved the desired level of damping after the cut-off frequency (i.e., a drop of 40 dB for a second-order filter). This was verified using the following equation:

$$d = \begin{cases} 40\,dB - damping\ , damping < 40\,dB \\ 0, damping \geq 40\,dB \end{cases} \quad (4)$$

A graphical representation of these four criteria is shown in Figure 7.

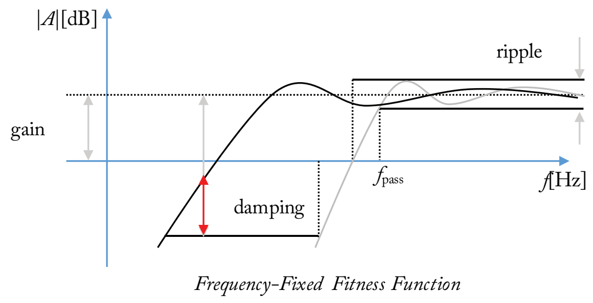

Figure 7: Multi-criteria fitness function components as shown in [5].

In the end, we combined the four characteristics into a single cost function:

$$\text{cost} = w_1 r + w_2 d + w_3 f_{\text{off}} + w_4 g \quad (5)$$

The four weights (w_1 to w_4) allow us to select which of the characteristics is more important to us. For example, if we favor achieving the desired level of damping and don't care so much about hitting the filter frequency precisely, we raise the value of w_2 and decrease the value of w_3. During our experiments, we emphasized gain and ripple since this produced the best results. We selected the weights experimentally by using values from 1 to 20 and then chose the set that produced the best results in several runs. The four weights were set to 15, 10, 5, and 4.

The new fitness function also simplified our algorithm in two important ways: (i) We do not need a comparison circuit anymore (which means less requirements for prior knowledge) and (ii) The evaluations of the four characteristics could be done automatically during the PyOpus simulation, thus reducing the amount of post-processing. This resulted in a noticeable speed increase during the test runs.

We soon discovered that the new fitness function finds workable circuits a lot more frequently (practically always) and works a lot more consistently during runs. Thus, we can conclude that it is crucial for a complex circuit to have a complex fitness function in order to be able to generate results consistently.

An example of a generated transfer function (compared to the idealized transfer function) is shown in Figure 8 with the matching generated circuit in Figure 9.

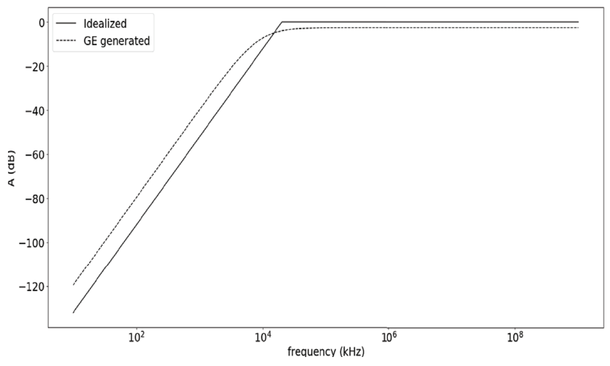

Figure 8: A second-order high-pass filter transfer function.

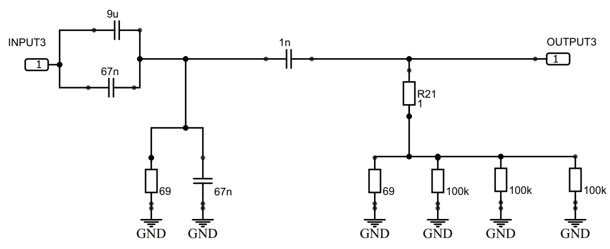

Figure 9: A second-order high-pass filter circuit generated by our GE algorithm.

We were also able to generate a low-pass filter with slight modifications to the cost function or, to be more precise, the PyOpus simulation parameters. Namely, we used the PyOpus measurement module to extract the cut-off frequency using the following expression:
m.ACbandwidth(abs(v('out')),abs(scale()),filter='hp'

To design a low-pass filter we simply switched the filter parameter to 'lp' and were able to proceed. The resulting transfer function and circuit are shown in Figures 10 and 11.

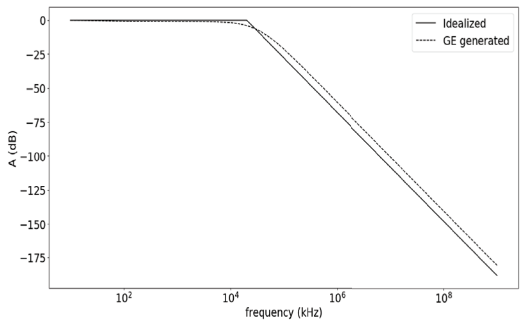

Figure 10: A second-order low-pass filter transfer function.

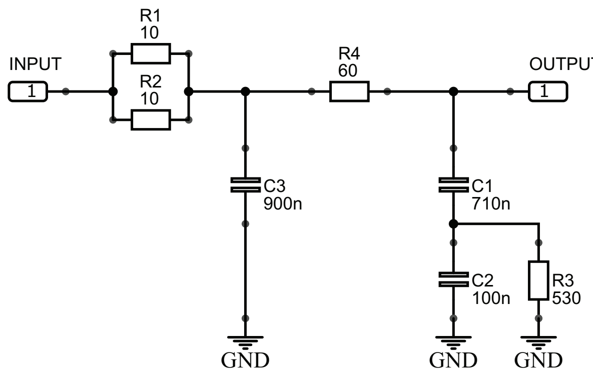

Figure 11: A second-order low-pass filter circuit generated using our method.

Looking at the circuits, we can see that there is a certain level of redundancy (for example two parallel resistors R1 and R2 in Figure 11), but (in our view) still not something to worry about. As already mentioned, much of this can be removed using a post processing algorithm that analyzes and optimizes the final netlist. This is, however, not possible while the algorithm is running, since it would require an extensive reworking of the chromosome structure. Nevertheless, we will consider this as a part of possible future improvements of our algorithm.

3.3 Third-order filters

For the last case study, we decided to increase the circuit complexity by increasing the level of the filter from second to third order. The implementation of such a change was extremely easy, since it only took us to change the target value of the damping factor from 40 dB to 60 dB in equation 3. The rest of the experiment used the same parameters (i.e., the grammar rules remained unchanged, no additional elements were added, and the number of runs and generations remained the same).

We were again able to consistently generate filter circuits with the desired characteristics in most of the runs. An example transfer function that we obtained from one of the evolution runs can be seen in Figure 12. This function belongs to the circuit shown in Figure 13. Interestingly enough, we did not need to add additional components into the algorithm during the initialization phase (meaning that we were able to create a third-order filter using up to 14 components).

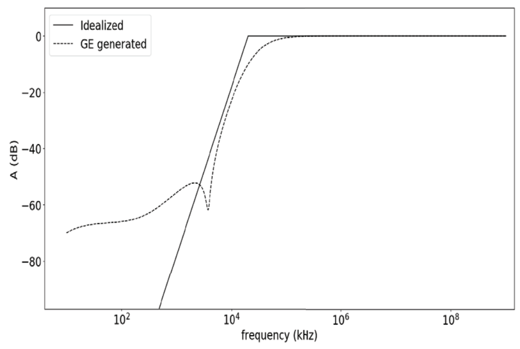

Figure 12: A third-order high-pass notch filter transfer function.

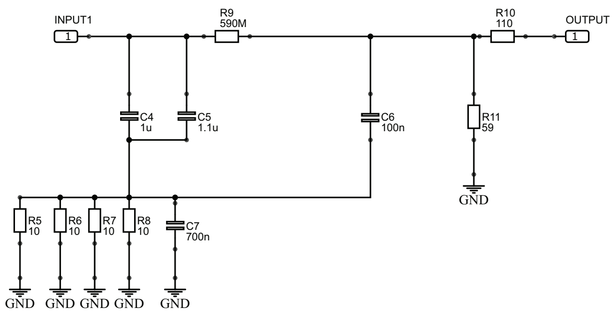

Figure 13: A generated third-order high-pass filter circuit.

We are able to make the same observations about the obtained circuits as we were during the previous case study – there is a certain level of redundancy (and bloat) but, due to an upper limit on the number of components, this remains on a manageable level and can be further reduced during post-processing.

4 Comparison with other methods

Compared to the original genetic programming based approach proposed by Koza [10], our approach offers more flexibility since it is not limited by the types of embryonic circuits introduced in the initialization phase. This means that we do not need to specify any starting topology or/and circuit and can leave the algorithm to find its own solution. This also reduces the amount of prior knowledge required to use our approach.

The approach presented in this paper also builds on findings of Castejon et al. [6] and Rojec et al. [5]. The former research group also used a GE approach and created separate rule sets similar to the ones presented in our work. They did not, however, limit bloat in the circuit (they allowed any number of elements) nor did they tackle more complex circuit examples. The latter is the consequence of them using only a very rudimentary cost function which, as we have demonstrated in this article, severely reduces the algorithm's success rate. In our approach, we used a multi-criteria cost function (similar to the one used in [5]) and were consequently able to produce more complex circuits as well.

An additional improvement made by our approach is a considerable increase in computation speed with which we generate the circuits. While Castejon et al. do not explicitly state the amount of time required for their experiments, we can learn that the approach used by Rojec et al. takes anywhere from one to 12 hours. All of the case studies presented in this work took less than 15 minutes per run to complete, while getting a comparable circuits. We could probably reduce this further by using multiple processors and hyper threading but since the process already took such a small amount of time, we left this for future work.

5 Conclusions

We successfully developed a GE based system for automated topology synthesis that works with a high-level rule set and a complex (or simple) fitness function. We were able to generate several circuits in a small amount of time with appropriate grammar modification. As a consequence we believe that this approach shows merit and can be of benefit to other engineers. It can also be developed further to improve its performance even more.

An additional improvement that we plan to develop is automatic post-processing of the evolved circuits. At the moment, we are only able to make sure that the evolved netlist contains correct component names and does not contain any illegal connections. This could be further improved by automatically detecting and removing any redundancies (e.g., replacing two or more serial or parallel elements of the same type with a single one).

Another option would be a repairing mechanism that would be used before the fitness function evaluation. Such a mechanism could detect faulty circuits, useless circuit branches and other defects even before evaluation and either try to correct them or flag the circuit as faulty and eliminate the individual. This could significantly improve the approach, but will require some time to develop since we would also have to modify the individuals' chromosome sequence to reflect the repairs.

We believe there lies much more potential for the application of the presented GE technique for an efficient evolution of useful and complex circuits than the science has been able to unearth so far. We will therefore aim to further develop the approach by increasing the complexity of the generated circuits, expanding the rule sets to include additional elements (transistors, amplifiers, etc.) and experiment with different options offered by the PyOpus environment (i.e., alternative modes of evaluation of the fitness function, parallelization, and others). Last but not least, we plan to work towards creating an open-source library to be available for other researchers and research groups in the community as a part of the PyOpus package.

6 Acknowledgments

The authors acknowledge the financial support from the Slovenian Research Agency (research core funding No. P2-0246 Algorithms and Optimization Methods in Telecommunications).

7 Conflicts of interest

The authors declare no conflict of interest.

The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

8 References

1. L. W. Nagel and D. O. Pederson, "SPICE (Simulation Program with Integrated Circuit Emphasis)," 1973.
2. J. Olenšek, T. Tuma, J. Puhan and Á. Bűrmen, "A New Asynchronous Parallel Global Optimization Method Based on Simulated Annealing and Differential Evolution," Applied Soft Computing, vol. 11, pp. 1481-1489, 2011.
3. Á. Bűrmen, F. Bratkovič, J. Puhan, I. Fajfar and T. Tuma, «Extended global convergence framework for unconstrained optimization,» Acta mathematica Sinica, vol. 20, pp. 433-440, 2004.
4. Á. Bűrmen, T. Tuma and I. Fajfar, «A combined simplex-trust-region method for analog circuit

optimization,» Journal of circuits, systems, and computers, vol. 17, pp. 123-140, 2008.
5. Ž. Rojec, Á. Bűrmen and I. Fajfar, «Analog circuit topology synthesis by means of evolutionary computation,» Engineering Applications of Artificial Intelligence, vol. 80, pp. 48-65, 2019.
6. E. Castejon and F. J. Carmona, "Automatic design of analog electronic circuits using grammatical evolution," Applied Soft Computing, vol. 62, pp. 1003-1018, 2018.
7. 8. G. Gielen and R. Rutenbar, Computer-aided design of analog and mixed-signal integrated circuits, New York: John Wiley & Sons , 2002.
8. R. Zebulum, M. Pacheco and M. Vellasco, "Comparison of different evolutionary methodologies applied to electronic filter design," in IEEE International Conference on Evolutionary Computation Proceedings, 1998.
9. J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, Cambridge, MA: MIT Press, 1992.
10. J. R. Koza, I. F. H. Bennett, D. Andre, M. A. Keane and F. Dunlap, "Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming," Trans. Evol. Comp, vol. 1, pp. 109-128, Jul 1997.
11. G. Györök, "Crossbar network for automatic analog circuit synthesis," in 2014 IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2014.
12. Z. Gan, Z. Yang, T. Shang, T. Yu and M. Jiang, "Automated synthesis of passive analog filters using graph representation," Expert Systems with Applications, vol. 37, no. 3, pp. 1887-1898, 2010.
13. L. Torres-Papaqui, D. Torres-Munoz and T.-C. E., "Synthesis of VFs and CFs by manipulation of generic cells," Analog Integr. Circuits Signal Process, vol. 46, pp. 99-102, 2006.
14. A. Bűrmen, J. Puhan, J. Olenšek, G. Cijan and T. Tuma, "PyOPUS - Simulation, Optimization, and Design," EDA Laboratory, Faculty of Electrical Engineering, University of Ljubljana, 2016.
15. M. O'Neill and C. Ryan, "Grammatical evolution," IEEE Transactions on Evolutionary, vol. 5, pp. 349-358, 2001.
16. I. Fajfar, Á. Bűrmen and J. Puhan, "Grammatical evolution as a hyper-heuristic to evolve deterministic real-valued optimization algorithms," Genetic programming and evolvable machines, vol. 19, pp. 473-504, 2018.
17. I. Fajfar, J. Puhan and Á. Bűrmen, "Evolving a Nelder–Mead Algorithm for Optimization with Genetic Programming," Evolutionary Computation, vol. 5, no. 3, pp. 351-373, 2017.
18. I. Fajfar and T. Tuma, "Creation of numerical constants in robust gene expression programming," Entropy, vol. 20, pp. 1-15, 2018.
19. M. Kunaver and T. Požrl, "Diversity in recommender systems - a survey," Knowledge-based systems, vol. 123, pp. 154-162, 2017.

Arrived: 22. 11. 2019
Accepted: 08. 01. 2020