Automatic Design of Analog Electronic Circuits using Grammatical Evolution

Federico Castejón[a], Enrique J. Carmona[a,*]

[a] *Departamento de Inteligencia Artificial, ETS Ingeniería Informática, Universidad Nacional de Educación a Distancia (UNED), Juan del Rosal 16, 28040, Madrid, Spain.*

Abstract

A new approach for automatic synthesis of analog electronic circuits based on Grammatical Evolution is presented. Grammatical Evolution is an evolutionary algorithm based on grammar which can generate code in any programming language and uses variable length linear binary strings. The decoding of each chromosome determines which production rules in a Backus-Naur Form grammar definition are used in a genotype-to-phenotype mapping process. In our method, decoding focuses on obtaining circuit netlists. A new grammar for generating such netlists and a variant of the XOSites-based crossover operator are also presented. A post-processing stage is needed to adapt the decoded netlist prior its evaluation using the NGSpice simulator. Our approach was applied to several case studies, comprising a total of seven benchmark circuits. A comparison with previous works in the literature shows that our method produces competitive circuits in relation to the degree of compliance with the output specifications, the number of components and the number of evaluations used in the evolutionary process.

Keywords: Automatic Circuit Design, Analog circuits, Evolutionary Electronics, Grammatical Evolution, NGSpice

1. Introduction

Since the late 1970s, analog circuits have been gradually replaced by digital circuits, but some functions still have to remain analog, mainly because transducers are analog as well. Unlike digital circuit design, analog circuit design still suffers from a lack of automatic design tools. Much effort has been made in Electronic Design Automation (EDA) to obtain help tools for the design of analog or mixed signal integrated circuits such as standard cell libraries or rule based expert systems [1]. Additionally, the field of Evolutionary Electronics appeared in 1998 [2], and since then, its goal has been the synthesis of electronic circuits using evolutionary algorithms. Inspired by natural selection, evolutionary algorithms can obtain solutions to complex problems. Thus, using a tentative population of potential circuit solutions, the best solutions are selected, crossed over and mutated. Successive generations lead to progressive improvement in the population in general and the best solution in particular. Evolutionary algorithms applied to synthesis tasks use neither design rules nor expert knowledge for the design [3]. That is why they can lead to unconventional solutions which challenge human designer intuition [4, 5].

Automatic circuit synthesis comprises two tasks: topology selection and circuit sizing. These tasks can be accomplished together or separately [6]. Thus, in some cases, evolutionary algorithms have only been used for circuit sizing, while topology selection has been done by human designers [7], or aided by aforementioned EDA tools [8]. In other cases, both tasks, topology selection and sizing, have been tackled simultaneously by evolutionary algorithms and, in general, by Evolutionary Electronics [2].

*Corresponding author

Email addresses: `federico.castejon@seap.minhap.es` (Federico Castejón), `ecarmona@dia.uned.es` (Enrique J. Carmona)

Evolutionary algorithms have been used successfully for the design of digital circuits [9, 10, 11, 12, 13]. On the other hand, they have also been used with good results for automatic synthesis of analog circuits [14, 15, 16, 17]. In the last case, Koza's group has been one of the most active in this area and its works based on genetic programming are especially relevant [18]. In particular, GP (and its variants) is considered the evolutionary paradigm with more successful results in the field of analog circuit design [19] and possibly remains so in the future [5, 20]. The scope of this work will be limited to the design of analog circuits. Thus, from now on, only this kind of circuits will be considered.

One of the main problems to be tackled in Evolutionary Electronics is how to encode a circuit. A circuit instance, which can be seen as a graph where edges are circuit components and vertices are circuit nodes, has to be encoded in a suitable chromosome structure. Next, a taxonomy based on [21] presents different types of approaches that are grouped according to how a circuit is encoded:

1. *Direct encoding*: components, node numbers and component values are encoded in a linear chromosome. In this case, the chromosome can have a fixed [22] or variable length [23, 24]. Alternatively, only components and node numbers are encoded in a linear chromosome, while component values are adjusted by a numerical optimizer [25, 26, 3]. In all cases, the decoding process is straightforward, leading to a circuit netlist.
2. *Encoding by grammar*: the chromosome is decoded into an expression whose syntax can be described by a grammar. Development expressions can be seen as part of this approach. A development expression defines the sequence of specialized functions which transform an initial circuit, normally called *embryo*, into a final circuit [27, 28, 18, 29, 30]. These approaches are based on genetic programming, using tree chromosomes as coding structures. Another way of encoding by grammar is using grammatical evolution, where linear binary chromosomes are used as coding structures and a set of sub-circuit development expressions contained in a grammar are used to decode each chromosome into a circuit [31].
3. *Encoding implicit interactions between components*: inspired by biologic genetic regulatory networks (GRN), this type of approaches encodes components and their connections with other components in a linear chromosome. Genes have regulatory regions which are influenced by other gene coding regions. These interactions determine the actual influence and final weight of the connections between the components [32, 21].

A new approach for designing analog electronic circuits based on Grammatical Evolution is presented here. This type of evolutionary paradigm can generate code in any programming language and uses variable length linear strings as chromosomes [33]. The chromosome values determine which production rules in a Backus-Naur Form (BNF) grammar definition are used in a genotype to phenotype mapping process in order to obtain a program or, as in our case, a circuit netlist. As far as the authors know, this paper, along with [31], also signed by the authors of this work, are the only two works in the literature that have applied GE to analog circuit design.

The main contribution of this work is twofold. First, we show evidence about how grammatical evolution can be used successfully for the automatic design of analog circuits. Second, unlike the grammar described in [31], which is oriented to the building of sub-circuit development expressions, the new grammar here presented focuses on generating circuit netlists.

The rest of the paper is structured as follows: Section 2 describes the problem to solve. Section 3 describes our method in detail. Section 4 focuses on applying our method to several case studies. Section 5 shows the results of the case studies. Next, a result discussion and a comparison with previous works in the field are presented in Section 6. The conclusions of our work are given in Section 7. A final appendix contains the topology and sizing of the best circuits obtained by our algorithm.

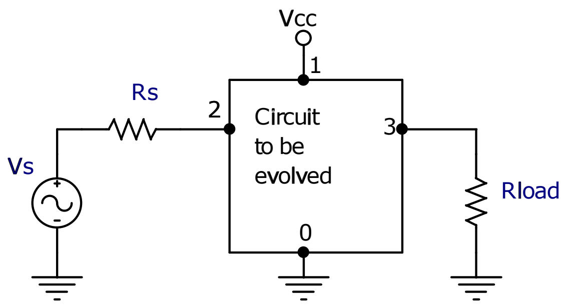


Figure 1: Example of a test fixture.

2. Problem Description

The global goal in this work is to design a circuit which meets a set of specifications. The circuit will also be constrained to use a set of components called *test fixture*. The test fixture normally consists of a positive power supply, an output load resistor and ground. Besides, depending on the problem, an input signal generator or a negative power supply, with their respective series resistors, may be needed. Fig. 1 shows an example of test fixture that uses five nodes. Four of them (0, 1, 2, and 3) are denominated *accessible nodes* because they can be used by the evolved circuit. In other case, they are denominated *inaccessible nodes* (e.g. the node linking R_s and V_s). The evolutionary circuit will be defined by a *netlist*, that is, a list of components and their connection nodes.

We assume that the set of specifications describes the desired output of the circuit in function of the input signal. In this way, it is possible to define an error function whose objective is to compare the desired output and the measured output of the proposed circuit as a solution. Therefore, the problem of automatically designing a circuit that meets a set of specifications can be transformed into a new optimization problem consisting of finding a circuit that minimizes the value returned by the mentioned error function as indicated in (1), where O is the desired output for the input signal, I, and $\widetilde{O}_j$ is the measured output for the j-th circuit when I is applied.

$$\underset{j}{minimize} \quad error(O - \widetilde{O}_j) \tag{1}$$

3. Method Description

In this section, a new proposal for automatic design of analog electronic circuits is presented. Several aspects related to our approach are described: (a) a short introduction to Grammatical Evolution (paradigm on which our method is based); (b) the grammar designed for decoding a chromosome into a netlist; (c) the taxonomy of types of chromosomes used in our approach; (d) the post-processing stage which is necessary to adapt the evolved netlist to the netlist format required by the simulator; and (e) the search engine used for our method. Finally, a decoding example is also shown.

3.1. Grammatical Evolution

Grammatical Evolution (GE) was introduced by Michael O'Neill and Connor Ryan [33]. GE is an evolutionary paradigm which can generate code in any programming language. It uses variable length linear binary strings as chromosomes and the decoding of each individual is based on a BNF grammar. Currently, GE is one of the most widely applied grammar-based approaches [34].

On the other hand, Genetic Programming (GP) uses parse trees as chromosomes and needs special crossover and mutation operators to work on this kind of chromosomal structures. These operators also have to preserve the closure property. These constraints do not apply in GE, where standard variation operators can be used and the decoding process based on grammar guarantees the correctness of the resulting program.

In GE, the unit of information is called *codon* and, normally, corresponds to one byte in the chromosome. Therefore, a codon can have 256 possible values. The decoding of a chromosome consists of traversing it from left to right, using each codon to choose an appropriate production rule according to the mapping function expressed in (2), where MOD is the modulus function and NR is the number of rules for the current non-terminal. Note that the output of MOD belongs to the set $\{0, 1, ..., (NR - 1)\}$. Therefore, the rules for each non-terminal symbol are numbered from 0 to $NR - 1$.

$$rule = codon\_value\ MOD\ NR \tag{2}$$

Additionally, if all the codons of a chromosome are completely read, but the expression is not fully decoded, then the reading will restart at the beginning of the chromosome and the decoding process continues. This mechanism is known as *wrapping* [33]. The wrapping parameter defines how many times the chromosome can be read before giving up.

3.2. Block grammar

In order to generate netlists of analog circuits, a suitable grammar has been developed. It is a BNF grammar, which is defined by a tuple of four elements: $\{S, T, N, R\}$ where S is the start symbol, T is the set of terminal symbols, N is the set of non-terminals and R is the set of production rules. A non-terminal symbol is a symbol that can be expanded into other non-terminal or terminal symbols, and a terminal symbol is a literal that cannot be further expanded. Applying the rules recursively to a string of symbols will usually terminate in a final output string consisting only of terminal symbols.

Table 1 shows the grammar used in our approach. It is expressed in Extended Backus Naur Form format [35] and comprises all the types of components that will be used in the case studies. This grammar always considers node #0 as ground.

The start symbol of the grammar is the LIST non-terminal, which is directly expanded as the COMPONENTS non-terminal. The COMPONENTS symbol can be expanded as one of the circuit components allowed or as one component plus the possibility to further expand the netlist (by adding a new component).

The term *block grammar* refers to the fact that the grammar is designed so that each component consumes a fixed length codon block. To achieve this, a non-terminal symbol, called DUMMY, is introduced to force the decoding process to consume as many codons as needed. This is implemented in the grammar by making a production rule for the DUMMY symbol with two options: “null1” and “null2”. Note that when a production rule has associated only one option, no codon is read [33]. Therefore, the assignation of two options for the DUMMY symbol forces to read and use a codon when this non-terminal symbol is being expanded. The null terminals produced by this rule will be removed later from the evolved netlist in a post-processing stage. Thus, DUMMY non-terminals are used as padding codons to build components with a fixed size block. Specifically, the grammar shown in table 1 uses 8-codon long blocks.

Node numbers are generated by the NODE non-terminal production rule. A special parameter of the grammar, denoted by MNN, defines the *maximum node number* that can be used by the circuit to evolve, without counting the *test fixture accessible node number* (TFANN). The MNN has to be set in advance by the user and the TFANN depends on the design specifications.

RESISTORs and CAPACITORs have a value expressed in scientific notation, $m \cdot 10^e$, where $1 \leq m < 10$ and the magnitude order range of the exponent, e, depends on the type of component (resistor or capacitor). TRANSISTORs can be BJT or MOSFET. Two types of BJT and MOSFET transistors are defined by the BJTTYPE and MOSTYPE non-terminals, respectively. If MOSFET transistors are used, it will be assumed that the substrate pins of PMOS and NMOS

Table 1: Generic grammar for netlist generation (see Section 3.2 for a detailed description)

S =	LIST
T =	{ “R”, “C”, “Q”, “M”, “QNPN”, “QPNP”, “null1”, “null2”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”, “e”, “.”, end-of-line character}
N =	{ LIST, LINE, RESISTOR, CAPACITOR, BJT, MOSFET, BJTTYPE, MOSTYPE, MODELNMOS, MODELPMOS, DUMMY, NODE, RESISTORVAL, CAPACITORVAL, CHANELWIDTH, DIGIT, NONZERODIGIT, EXPONENT, EOL }
R =	*comprises the following rules of production*
LIST =	COMPONENTS;
COMPONENTS =	RESISTOR | RESISTOR, COMPONENTS | CAPACITOR | CAPACITOR, COMPONENTS | BJT | BJT, COMPONENTS | MOSFET | MOSFET, COMPONENTS;
RESISTOR=	"R", NODE, NODE, DUMMY, RESISTORVAL, DUMMY, EOL;
CAPACITOR =	"C", NODE, NODE, DUMMY, CAPACITORVAL, DUMMY, EOL;
BJT =	"Q", NODE, NODE, NODE, BJTTYPE, DUMMY, DUMMY, DUMMY, EOL;
MOSFET =	"M", NODE, NODE, NODE, MOSTYPE, CHANELWIDTH, EOL;
BJTTYPE =	"QNPN" | "QPNP";
MOSTYPE =	MODELNMOS | MODELPMOS;
MODELNMOS =	"0", "NMOS1 L=10u W=”
MODELPMOS =	"1", "PMOS1 L=10u W=”
DUMMY =	"null1" | "null2";
NODE =	"0" | "1" | "2" | "3" | "4" | "5" | ... | “TFANN+MNN-1”;
RESISTORVAL =	NONZERODIGIT, “.”, DIGIT, “e”, DIGIT;
CAPACITORVAL =	NONZERODIGIT, “.”, DIGIT, “e”, EXPONENT;
CHANELWIDTH =	NONZERODIGIT, DIGIT, "u" | "1", DIGIT, DIGIT, "u";
DIGIT =	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
NONZERODIGIT =	"1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
EXPONENT =	"-12" | "-11" | "-10" | "-9" | "-8" | "-7" | "-6" | "-5" | "-4" | "-3";
EOL	*end-of-line character*

are connected to the maximum and minimum voltage nodes, respectively. These pins are coded in the MODELPMOS and MODELNMOS non-terminals. MOSFET channel length is always fixed to $10\mu m$, while channel width can be evolved between $10\mu m$ and $199\mu m$, in the same way as in [18].

The grammar shown in table 1 is generic. Therefore, it should be adjusted to each specific design problem. For example, if a type of component is not used because it is not included in the design specifications, it should be removed from the final grammar. On the other hand, other components could be added to the grammar such as, for example, diodes or inductors. A chromosome decoding example using the grammar described here will be presented in subsection 3.5.

3.3. Managing different types of chromosomes

In GE, there could be chromosomes which cannot be mapped to a final expression. This happens when a chromosome is read as many times as the value assigned to the wrapping parameter, without producing an expression free of non-terminal elements [33]. In our case, these chromosomes, which can be called *inexpressible chromosomes*, do not produce a netlist and therefore cannot be simulated. They will be penalized strongly in order to have a low probability of being chosen during the parent selection stage. In particular, since lower fitness values are better (minimization problem), penalization can be done by assigning a high value as the fitness value and arbitrarily fixed to 5×10^7. This value is the maximum fitness allowed, so any other fitness value will be lower than this one. In this context, an inexpressible individual will only win a tournament, that is, it will be selected as parent, if all individuals randomly selected to compete in the tournament are inexpressible.

Additionally, badly formed circuits can also appear after decoding. This occurs when the evolved circuit has no connection with at least one of the test fixture accessible nodes such as the power supply, the load resistor or the signal generator. These circuits are unfeasible and their chromosomes will be called *unfeasible chromosomes*. This kind of chromosomes is less penalized than inexpressible ones, because it is preferable to obtain chromosomes that generate a netlist although they are unfeasible. In this case, the penalization value is arbitrarily fixed to 2.5×10^7, that is, half the penalization value used for inexpressible chromosomes.

There also exists the problem of chromosomes generating a netlist with one or more dangling components, that is, components with one or more pins connected to an isolated node. This case is not considered as an unfeasible circuit, since it can be easily tackled. There are several approaches to manage dangling components:

1. Remove the dangling component [36, 18].
2. Connect the dangling component to a preexisting circuit node (ground, positive or negative power supply, input, output, etc.) [37]
3. Insert a high valued resistor between the ground and dangling node [21].

The first and second approaches can be complicated to implement when the dangling component has three or more pins. Thus, our method uses the third approach. Finally, the chromosomes which are not classified in the above categories will be denominated *feasible chromosomes*.

3.4. Netlist post-processing

The netlist just generated in the decoding process cannot be directly fed to the simulator. A post-processing step is needed, which does the following actions in the evolved netlist:

1. The test fixture netlist, which depends on the specifications of the circuit to be designed, is added.
2. If the evolved circuit is not completely connected to all the test fixture accessible nodes, the netlist is marked as *unfeasible*, penalized accordingly and is not simulated.
3. The "null1" and "null2" symbols are deleted (see Section 3.2).
4. Additional resistors are added to connect dangling components. A $1G\Omega$ resistor is connected between the ground and each dangling node.
5. The short-circuited components are removed because they do not have any function in the circuit.
6. The component labels are numbered because the simulator needs to identify each component with a unique label and automatic component numbering cannot be modeled by a context-free grammar.
7. The component values generated in scientific notation are converted to standard notation, and unit prefixes are also added.

Finally, the resulting netlist is simulated. Here NGSpice [38], which is based on Spice3 [39], is used for this task. Spice and derived software is industry's de facto standard electronic circuit simulation software, and even referred as "gold standard" [40].

The NGSpice output of a candidate circuit is processed to produce the resulting signal (i.e. output voltage or output current) which is compared with the design specifications in order to obtain the fitness value for that circuit. If an NGSpice error is obtained when a circuit is simulated, then this circuit is labeled as unfeasible and penalized accordingly.

3.5. Decoding example

In this example, the design problem consists of obtaining a BJT-based amplifier, considering a set of specifications and including the test fixture shown in Fig. 1. The MNN parameter is set by the user to value 6 and, according to the aforementioned figure, $TFANN = 4$. Therefore, the total number of nodes that the circuit has to evolve is 10. A candidate chromosome could be the following, {53, 34, 22, 60, 4, 122, 32, 71, 9, 251, 74, 7, 82, 140, 93, 232, 66, 94, 33, 102, 28, 40, 63, 67, 255, 3, 45, 32}. The decoding process, using the grammar shown in table 1, is as follows:

1. The decoding starts with the S symbol, that is, with LIST.
2. LIST is expanded as COMPONENTS. Since the production rule implies no choices, no codon is drawn.
3. A codon value is needed for the expansion of COMPONENTS and the first codon is 53, so $53\, MOD\, 8$ selects rule *#5* which gives: **BJT, COMPONENTS**.
4. BJT has only one associated rule. Therefore it is expanded directly and the result is: Q, **NODE, NODE, NODE, BJTTYPE, DUMMY, DUMMY, DUMMY, EOL, COMPONENTS**.
5. Q is a terminal symbol and does not consume any codons. Next, the expansion of the rule for the NODE non-terminal with codon value 34 selects rule *#4* ($34\, MOD\, 10$), which is associated with node number 4: Q 4, **NODE, NODE, BJTTYPE, DUMMY, DUMMY, DUMMY, EOL, COMPONENTS**.
6. The expansion of the other two nodes, with codon values 22 and 60, gives node numbers 2 ($22\, MOD\, 10$) and 0 ($60\, MOD\, 10$), respectively: Q 4 2 0, **BJTTYPE, DUMMY, DUMMY, DUMMY, EOL, COMPONENTS**.
7. The expansion of rule for the BJTTYPE symbol with codon value 4 selects rule *#0* ($4\, MOD\, 2$)*,* which is associated with QNPN, that is, the type of BJT transistor*:* Q 4 2 0 QNPN, **DUMMY, DUMMY, DUMMY, EOL, COMPONENTS**.
8. The following three DUMMY non-terminals are used as padding codons in order to use an 8-codon fixed sized block. The expansion with codon values 122, 32 and 71 selects null1, null1 and null2, respectively: Q 4 2 0 QNPN null1 null1 null2 **EOL, COMPONENTS**.
9. EOL has only one rule and is expanded as an end-of-line character. At this point, the first component of the netlist is decoded completely and will not be shown in the next steps.
10. Next, the COMPONENTS non-terminal is expanded with codon value 9, selecting rule #1 which produces: **RESISTOR, COMPONENTS**.
11. The RESISTOR symbol has only one rule and does not consume any codons. Therefore, it is expanded and the result is: R, **NODE, NODE, DUMMY, RESISTORVAL, DUMMY, EOL, COMPONENTS**.
12. The R symbol is terminal and does not consume any codons. The NODE non-terminals with codon values 251 and 74 produce node numbers 1 and 4, respectively: R 1 4 **DUMMY, RESISTORVAL, DUMMY, EOL, COMPONENTS**.
13. The process continues in a similar way until the evolved netlist is obtained. In particular, the decoding process ends when the expression has no more non-terminals to expand. This happens when the codon 67 is decoded. Thus, in this case, the last four codons of the chromosome are not used.

The final evolved netlist is:

```
Q 4 2 0 QNPN null1 null1 null2
R 1 4 null2 1.0e3 null1
C 4 3 null1 1.0e-9 null2
```

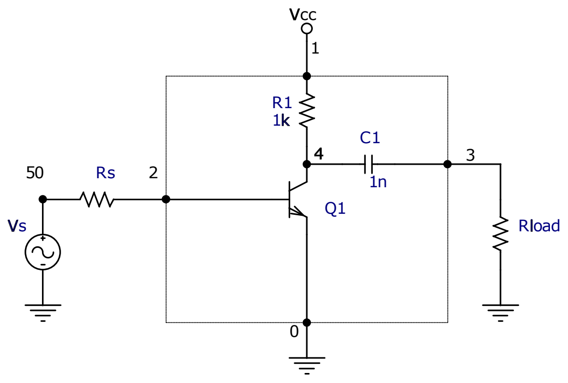


Figure 2: Example of evolved circuit connected to the test fixture of Fig. 1.

Next, the post-processing stage is applied, according to the steps outlined in Section 3.4. In this case, it comprises: (a) addition of the test fixture netlist (see Fig. 1), (b) deletion of null parameters, (c) component label numbering, and (d) addition of unit suffixes to component values. Those steps related to unfeasible circuit, short-circuited or dangling components do not apply in this case. The resulting netlist is as follows:

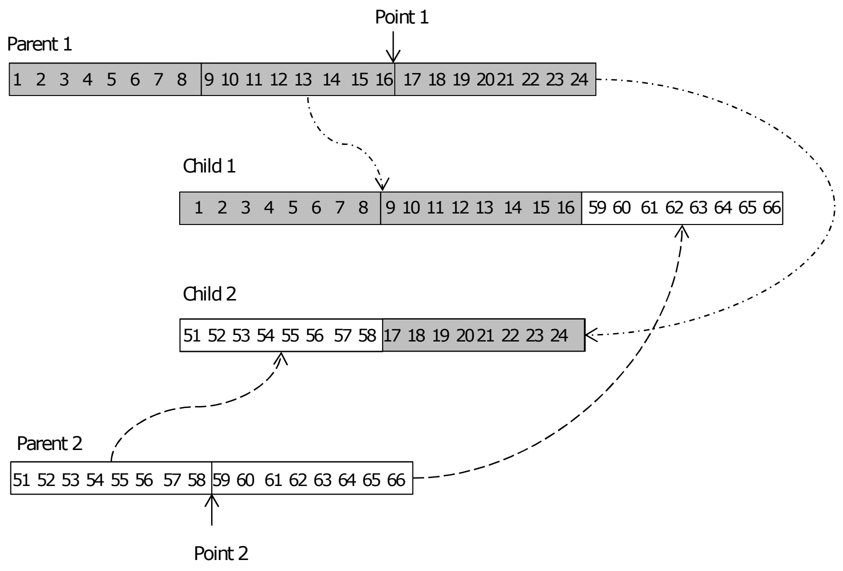
```
*Test fixture netlist
Vcc 1 0 dc 5
Vs 0 50 0.0 ac 0.001
Rs 50 2 100
Rload 3 0 100
*Evolved netlist
Q1 4 2 0 QNPN
R1 1 4 1.0k
C1 4 3 1.0n
```

Note that the node used to connect Vs and Rs is not an accessible node. However, it is necessary to label it in order to define completely the test fixture. To do that, any node number not belonging to the set $\{0, 1, ..., (TFANN + MNN - 1)\} \equiv \{0, 1, ..., 9\}$ is valid. In this case, the value 50 has been arbitrarily selected. The circuit associated with the final netlist is shown in Fig. 2.

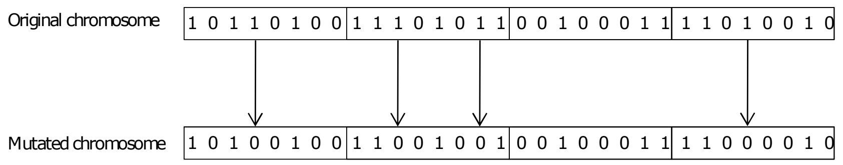
3.6. Search engine

It is known that GE is a modular evolutionary paradigm, that is, it deals separately with the chromosome decoding strategy and the solution search strategy. In particular, the search does not need to be carried out by a specific algorithm. This means that any existing evolutionary algorithm using integer strings as chromosomes can be used. In particular, genetic algorithms (GA) are normally used as a search engine in GE, although it is also possible to perform the search using other methods, such as algorithms based on particle swarm optimization. In our case, a GA-based approach is used. Here, linear chromosomes are composed of a variable length of codons (bytes).

We use a block crossover operator, named *one-block crossover*, which takes advantage of the block grammar. It is a variant of the crossover operator described in [41]. In particular, the crossover point is a multiple of the block size which is defined by the grammar (see Section 3.2). Thus, material is exchanged at a component boundary, as is shown in Fig. 3a. In this way, we avoid the destructive behavior associated with a classic crossover operator which could interchange parts of different components. This operator also includes a maximum chromosome length constraint for the children obtained. If the constraint is violated, the child will be



(a) One-block crossover operator



(b) Bitwise mutation operator

Figure 3: Operative followed by the variation operators implemented.

clipped to the maximum length. This constraint helps to control the negative effects of the well-known *bloat effect* [4]. In section 5, the benefits of the one-block operator are compared with a standard one-point operator. On the other hand, the mutation operator is a classic bitwise operator where bits are changed randomly according to a mutation rate, as it can be seen in Fig. 3b. The survivor selection is based on generational replacement, and the parent selection is implemented by 3-individual tournament . Finally, the fitness function is always defined in such a way that lower values correspond to better circuits, so the circuit design problem is tackled as a minimization problem.

4. Case Studies

Two sets of benchmark circuits were chosen to test the approach proposed. The first set, denoted by non-computing circuits, comprises a temperature sensing circuit, a voltage reference circuit and a Gaussian function generator circuit. The performance of the method proposed will be analyzed from a statistical point of view using this set of circuits. A second set, denoted by computational circuits, comprises four computing circuits: square, square root, cube and cube root. The best circuits obtained will be shown and their results will be assessed and compared with those of previous works [18, 21, 42, 43, 44, 24]. In this section, the configuration parameters of our method and the design specifications of the target circuits are presented.

4.1. Configuration Parameters

The netlist generation grammar has to be adjusted to the specific problem being tackled, bearing in mind the following items: component types allowed, test fixture accessible node number (TFANN), and maximum node number (MNN). Component types not used in the problem

Table 2: Parameter values of the GE-based method

Circuit type	Temperature sensor (TS), Voltage reference (VR), Gaussian function (GF) or Computing circuits (CC)
Grammar	Generic grammar (see table 1) adapted to benchmark circuit
Fitness function	Depends on benchmark circuit (see subsections 4.2 and 4.3)
Population size (μ)	$1,000$
Representation	Codon strings of variable length
Generations	$3,000$
Initialization	Random codon strings of $150-250$ length
Maximum chromosome length	294 (TS), 294 (VR), 336 (GF) and 294 (CC) codons
Crossover	One-block crossover operator (see section 3.6)
Crossover probability	0.5
Block size (component)	7 (TS), 7 (VR), 8 (GF) and 7 (CC) codons
Mutation	Bitwise
Mutation probability	0.001
Parent selection	Tournament selection (size=3)
Survivor selection	Generational replacement ($\lambda = \mu$)
Elitism	2
Wrapping	4
Termination condition	Maximum number of generations
Random number generator	Mersenne Twister
Number of runs per circuit	50

have to be removed from the grammar given in table 1, by modifying the COMPONENTS non-terminal and other related non-terminals. After removing one or several component types, it could be necessary to adjust the block size in the grammar. For example, if MOSFET transistors are not necessary, which are responsible for using 8-byte long blocks in the grammar of table 1, the block size of the adapted grammar can be reduced to 7 bytes, which is done by removing the last DUMMY non-terminals in the remaining component types. Note that the size of 7 bytes is imposed by the RESISTOR and CAPACITOR non-terminals. Table 2 shows the configuration parameters used in our method. The parameter values were adjusted experimentally from preliminary runs. The GA parameter adjustment was made testing different combinations of parameter values and evaluating each of them in function of the number of successful circuits obtained in the set of non-computational circuits. The best GA parameter configuration obtained with this class of circuits was maintained without changes in the set of computational circuits. In particular, in relation to the wrapping parameter, there is evidence in the literature that if wrapping is removed, a degradation of performances will be obtained [45]. However, there is no rule of thumb for optimum tuning of this parameter. The wrapping parameter was set to different values in preliminary runs, but there were no significant differences in the results. We left a final wrapping value of 4 in order to favor the emergence and evolution of feasible individuals in the first generations. Additionally, we found that wrapping was not much used in the final generations.

4.2. Specifications of non-computational circuits

All specifications described for the set of non-computational circuits are the same as those defined by Koza [18], including the definition of the different fitness functions and the type of components used.

4.2.1. Temperature sensing circuit

This circuit takes its own temperature as input and should provide an output voltage proportional to the temperature value. The range of temperature is $0°C \leq T \leq 100°C$ and the range of the output voltage should be $0V \leq V_o \leq 10V$. Candidate circuits will be simulated with a temperature sweep in the above mentioned range at intervals of $5°C$ giving 21 fitting points. When the absolute difference between the desired output voltage, V_{o_i}, and measured output voltage, $\widetilde{V}_{o_i}$, is less than or equal to $0.1V$, a hit is scored. A hit scored at a fitting point means that the measured output is close enough to the desired output at that point. For example, if a circuit (individual) gets a hit in 15 of the 21 fitting points, then its ratio of hits is said to be 15/21. Only if all available hits are scored, the circuit is considered successful. The fitness function is calculated using (3), where the weight factors, w_i, are calculated using (4), which depend on whether a hit is scored or not at the evaluated point.

$$fitness = \sum_i w_i |V_{o_i} - \widetilde{V}_{o_i}| \tag{3}$$

$$w_i = \begin{cases} 1.0 & when\ |V_{o_i} - \widetilde{V}_{o_i}| \leq 0.1V \\ 10.0 & when\ |V_{o_i} - \widetilde{V}_{o_i}| > 0.1V \end{cases} \tag{4}$$

The types of components allowed are 2N3904 and 2N3906 BJT transistors, resistors and capacitors. The fixture test has four accessible nodes: two power supplies ($+15V$ and $-15V$), one output load resistor ($1K\Omega$) and ground.

4.2.2. Voltage reference circuit

The goal of this circuit is to provide a fixed output voltage of $2V$ when the input voltage varies inside the interval $4V \leq V_i \leq 6V$ and the temperature varies in the interval $0°C \leq T \leq 100°C$.

Candidate circuits will be simulated with a voltage DC sweep in the first range at intervals of $0.1V$ and a temperature sweep in the second range at intervals of $25°C$ giving a total of 105 (21×5) fitting points. When the absolute difference between the desired output voltage, V_{o_i}, and measured output voltage, $\widetilde{V}_{o_i}$, is less than or equal to $0.02V$, a hit is scored. The fitness function is calculated using (5), where the weight factors, w_i, are evaluated using (6), which depend on whether a hit is scored or not at the evaluated point.

$$fitness = \sum_i w_i |V_{o_i} - \widetilde{V}_{o_i}| \tag{5}$$

$$w_i = \begin{cases} 1.0 & when\ |V_{o_i} - \widetilde{V}_{o_i}| \leq 0.02V \\ 10.0 & when\ |V_{o_i} - \widetilde{V}_{o_i}| > 0.02V \end{cases} \tag{6}$$

The types of components allowed are 2N3904 and 2N3906 BJT transistors and resistors. The test fixture has three accessible nodes: one power supply (with a $1K\Omega$ series resistor), one output load resistor ($10K\Omega$), and ground.

4.2.3. Gaussian function generator circuit

The goal of this circuit is to generate an output current which is a Gaussian function of the input voltage. The output current will be measured on an output voltage generator. This problem is credited to Adrian Stroica of JPL in Pasadena, California [18]. The input voltage can vary inside the interval $2V \leq V_i \leq 3V$ and Gaussian function will be centered in $2.5V$ with a standard deviation of $0.1V$. The output current peak will be of $80nA$. Candidate circuits will be simulated with a voltage DC sweep in the above mentioned range at intervals of $0.01V$ giving a total of 101 fitting points. When the absolute difference between the desired output current, I_{o_i}, and measured output current, $\widetilde{I}_{o_i}$, is less than or equal to $5nA$, a hit is scored. The fitness function is calculated using (7), where the weight factors, w_i, are calculated using (8), which depend on whether a hit is scored or not at the evaluated point.

$$fitness = \sum_i w_i |I_{o_i} - \widetilde{I}_{o_i}| \tag{7}$$

$$w_i = \begin{cases} 10^6 & when\ |I_{o_i} - \widetilde{I}_{o_i}| \leq 5nA \\ 10^7 & when\ |I_{o_i} - \widetilde{I}_{o_i}| > 5nA \end{cases} \tag{8}$$

The types of components allowed are N-channel and P-channel MOSFET transistors and resistors. The test fixture has four accessible nodes: one power supply ($+5V$), one input signal generator ($2-3V$ and a 1Ω series resistor), the above-mentioned output voltage generator, and ground.

4.3. Specifications of computational circuits

All specifications described for the set of computational circuits are the same as those defined in [18], including the definition of the different fitness functions and the type of components used. The set of computational circuits consists of four circuits which compute a mathematical function of the input voltage value. Four functions are used: square, square root, cubing and cube root. Input voltage varies from $-250mV$ to $250mV$, with the exception of the square root circuit which varies between $0V$ and $500mV$. Candidate circuits will be simulated with a voltage DC sweep in the above-mentioned ranges at intervals of $25mV$ giving a total of 21 fitting points. When the absolute difference between the desired output voltage, V_{o_i}, and measured output voltage, $\widetilde{V}_{o_i}$, is less than or equal to $1\%V_{o_i}$, a hit is scored. The fitness function is calculated using (9), where the weight factors w_i, are calculated using (10), which depend on whether a hit is scored or not at the evaluated point.

$$fitness = \sum_i w_i |V_{o_i} - \widetilde{V}_{o_i}| \tag{9}$$

$$w_i = \begin{cases} 1.0 & when\ |V_{o_i} - \widetilde{V}_{o_i}| \leq 1\%V_{o_i} \\ 10.0 & when\ |V_{o_i} - \widetilde{V}_{o_i}| > 1\%V_{o_i} \end{cases} \tag{10}$$

The types of components allowed are 2N3904 and 2N3906 BJT transistors and resistors. The test fixture has five accessible nodes: two power supplies ($+15V$ and $-15V$), one input signal generator (with a $1K\Omega$ series resistor), one output load resistor ($1K\Omega$), and ground.

5. Results and Evaluation

In this section, the results of the case studies are shown. In order to make a statistical evaluation of the method proposed, common performance measures of evolutionary algorithms such as the success rate (SR) and mean best fitness (MBF) are used [4]. Given an experiment comprising a number of test runs of the method, SR is defined as the ratio of the number of successful runs in relation to the total number of runs, where one success is obtained when the evolved circuit scores all the hits. On the other hand, MBF is the fitness average of the best individuals, each of which is obtained at the end of each run, whether successful or not. Finally, the minBF is also considered, that is, the minimum best fitness obtained in the successful runs, which corresponds to the best circuit designed by the method in a particular experiment (50 runs).

For the set of non-computational circuits, an analysis of two method parameters is done. In particular, the effects observed for the number of generations and the maximum number of nodes are shown. The outputs of the best circuits obtained for this set are also shown. The case study for the set of computational circuits only focuses on the circuit design, so only the outputs of the best circuits are shown.

The time for a run depends on the circuit evolved and the number of generations used. In any case, for $3,000$ generations, a run takes an average of 70-80 minutes in a cluster with 5 PCs. Each PC has a Core 2 Quad Intel CPU @ $2.66Ghz$, four cores, $4GB$ RAM and $250GB$ disk, making a total of 20 cores.

Table 3: Results obtained for different experiments (rows) in the set of non-computational circuits, depending on the maximum number of nodes (MNN). Each experiment corresponds to 50 runs and 3,000 generations per run. #Succ. is the number of successes, SR is the success rate, minBF is the minimum Best Fitness, and MBF is the Mean Best Fitness

Target	MNN	# Succ.	SR (%)	minBF	MBF±std	Hits avg±std / max
sensor	4	21	42.0	0.176	6.559 ± 11.196	19.2 ± 2.6 / 21
sensor	6	21	42.0	0.169	5.294 ± 6.526	19.5 ± 1.7 / 21
sensor	8	24	48.0	0.172	3.653 ± 4.256	19.9 ± 1.4 / 21
sensor	10	26	52.0	0.140	2.628 ± 2.942	20.2 ± 1.1 / 21
sensor	15	22	44.0	0.140	4.574 ± 5.326	19.7 ± 1.4 / 21
sensor	20	16	32.0	0.286	5.849 ± 7.007	19.3 ± 2.1 / 21
sensor	30	13	26.0	0.066	6.328 ± 8.598	19.3 ± 1.9 / 21
vref	4	1	2.0	0.773	33.756 ± 21.167	49.3 ± 27.4 / 105
vref	6	4	8.0	0.112	26.567 ± 16.228	56.5 ± 26.3 / 105
vref	8	5	10.0	0.183	18.145 ± 13.498	68.8 ± 23.3 / 105
vref	10	13	26.0	0.125	14.093 ± 15.096	78.0 ± 26.5 / 105
vref	15	6	12.0	0.231	20.429 ± 16.451	65.3 ± 26.1 / 105
vref	20	5	10.0	0.171	17.955 ± 17.072	71.5 ± 25.9 / 105
vref	30	6	12.0	0.309	18.255 ± 16.550	70.0 ± 27.5 / 105
gauss	4	6	12.0	0.078	10.632 ± 6.990	$69,6 \pm 15.2$ / 101
gauss	6	12	24.0	0.040	6.702 ± 6.782	78.1 ± 17.8 / 101
gauss	8	8	16.0	0.068	6.645 ± 6.372	77.4 ± 15.9 / 101
gauss	10	12	24.0	0.047	5.945 ± 6.276	78.9 ± 18.2 / 101
gauss	15	13	26.0	0.030	6.474 ± 6.682	79.8 ± 16.8 / 101
gauss	20	13	26.0	0.041	6.856 ± 7.230	80.3 ± 17.5 / 101
gauss	30	8	16.0	0.060	7.649 ± 6.674	74.9 ± 17.7 / 101

5.1. Set of non-computational circuits

The results on the set of non-computational circuits are shown in Table 3, where different values of the MNN parameter were tested. Due to the stochastic nature of an evolutionary algorithm, each experiment, represented in each row of the table, was repeated 50 runs. As it can be seen, the SR and MBF values obtained depend on the goal circuit, showing that some circuits are harder to tackle than others. However, for design problems, such as those discussed here, the most important challenge is to implement a method that creates one good solution at least once [4]. In this context, the minBF value obtained by our method is always competitive and, in the three cases, the best circuit obtained always reaches 100% of hits (considering 50 runs).

Fig. 4 shows graphically the effect of the MNN parameter on the SR for the three circuits studied. As it can be seen, there is a peak or plateau where the SR reaches the maximum value for each type of circuit. Specifically, in the temperature sensor and voltage reference circuits, there is a peak for $MNN = 10$ nodes and, in the Gaussian function generator, a plateau for $MNN = 15 - 20$ nodes. According to these results, the following strategy can be established in order to systematically tune the MNN parameter. Starting with a fixed and small MNN value, several runs are done and, for each of then, the value of the number of hits is analyzed. Then, the MNN value is gradually increased, until the number of hits of the evolved circuit reaches the maximum.

The number of components is also affected by the *MNN* parameter. Specifically, Fig. 5 shows the average number of components for the successful circuits obtained in each experiment versus MNN. For calculating the average number of components, it was used neither the test fixture

Table 4: Results obtained for different experiments (rows) in the set of non-computational circuits, depending on the number of generations. Each experiment uses an MNN of 6 and corresponds to 50 runs. #Succ. is the number of successes, SR is the success rate, minBF is the minimum Best Fitness and MBF is the Mean Best Fitness

Target	# Gen.	# Succ.	SR (%)	minBF	MBF±std	Hits avg±std / max
sensor	$10,000$	29	**58.0**	0.065	3.881 ± 5.936	19.9 ± 1.8 / 21
sensor	$3,000$	21	42.0	0.169	5.294 ± 6.526	19.5 ± 1.7 / 21
vref	$10,000$	7	**14.0**	0.188	17.353 ± 14.588	70.7 ± 25.0 / 105
vref	$3,000$	4	8.0	0.112	26.567 ± 16.228	56.5 ± 26.3 / 105
gauss	$10,000$	22	**44.0**	0.036	3.943 ± 5.167	85.0 ± 17.1 / 101
gauss	$3,000$	12	24.0	0.040	6.702 ± 6.782	78.1 ± 17.8 / 101

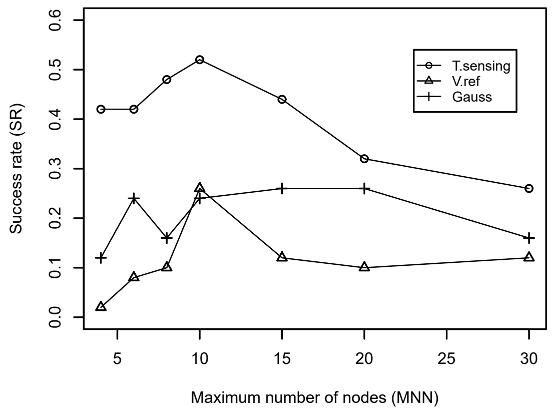


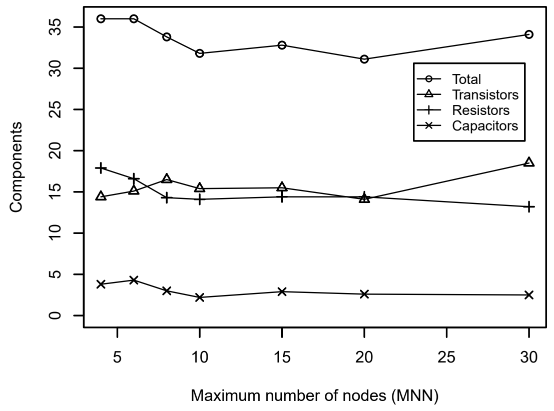
Figure 4: Success rate (SR) versus maximum number of nodes (*MNN*) for the set of non-computational circuits.

components nor simplifications based on changing parts of the evolved circuit by equivalent sub-circuits. The average number of components reaches a minimum for the MNN equal to 4 and 8 in the voltage reference and Gaussian function generator, respectively. On the other hand, in the case of the temperature sensor, a minimum quasi-plateau is observed for the MNN ranging from 10 to 20.

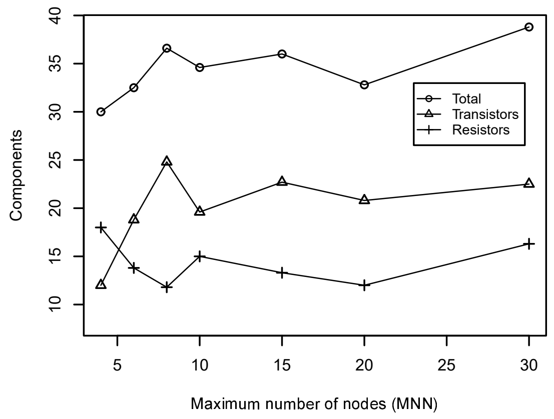
In addition, in order to study whether an improvement is still possible, the number of generations is increased. Table 4 shows the results obtained for $10,000$ generations. As it can be seen, the SR, MBF and minBF improved in relation to the study with $3,000$ generations.

Finally, the measured outputs of the best circuits are compared with the expected ones. Thus, Fig. 6a, 6b and 6c show, respectively, the output voltage of the best temperature sensing circuit, the output voltage of the best voltage reference circuit, and the output current of the best Gaussian function generator circuit. As it can be seen, a good fit is obtained. The topology and sizing of each one of these circuits are shown in the final appendix (see Fig. A.9, A.10 and A.11, respectively).

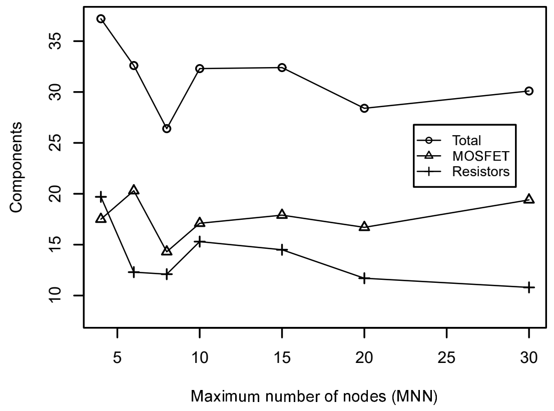
The introduction of the one-block crossover operator allows our method to interchange complete blocks (circuit components) between parent chromosomes. In contrast, a standard one-point operator could be more aggressive because it can break a chromosome at any point and swap information with different internal meaning. Table 5 shows a comparison between both operators. The study is performed for MNN values with high SR scores (see fig. 4). For each of the three circuits, the SR value obtained by the one-block operator is greater than that one of



(a) Temperature sensing.

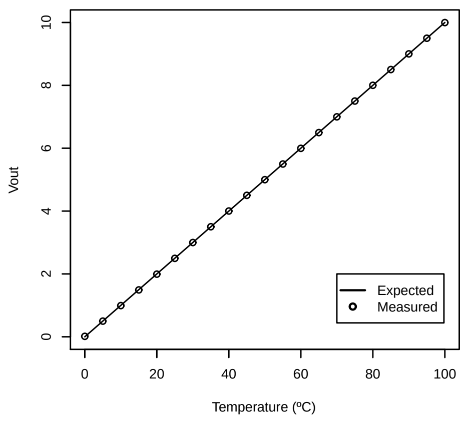


(b) Voltage reference.

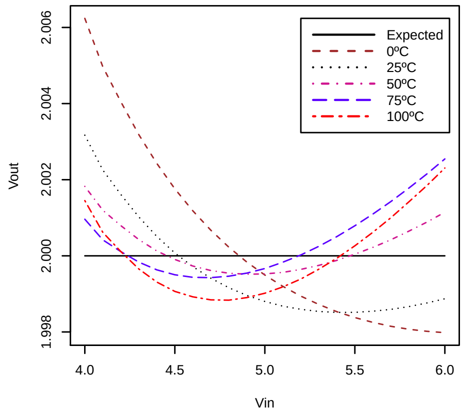


(c) Gaussian function generator.

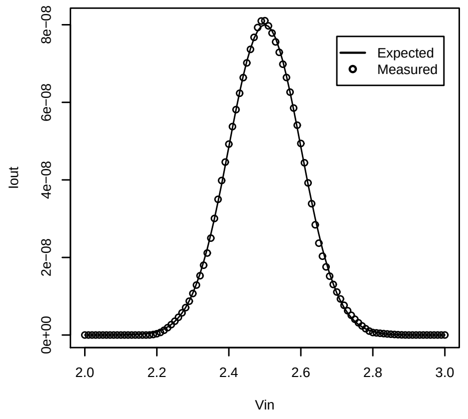
Figure 5: Mean number of components versus maximum number of nodes (MNN) for the set of non-computational circuits.



(a) Temperature sensing

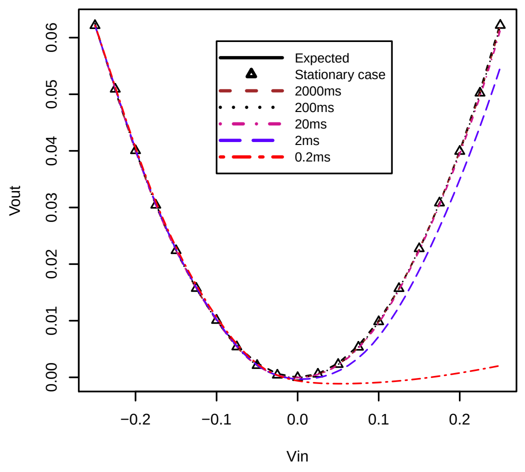


(b) Voltage reference. Note the reduced scale of Y axis ($\triangle V_{out} = 2mV$)

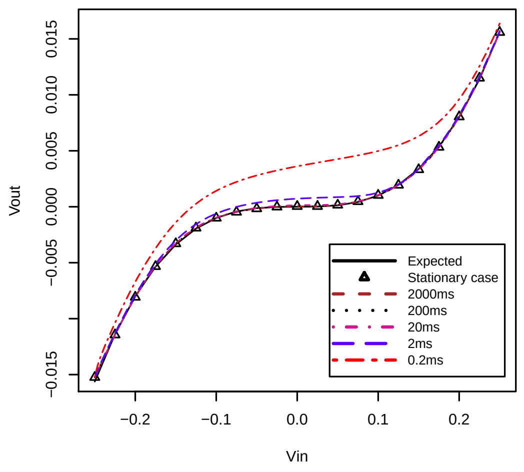


(c) Gaussian function generator

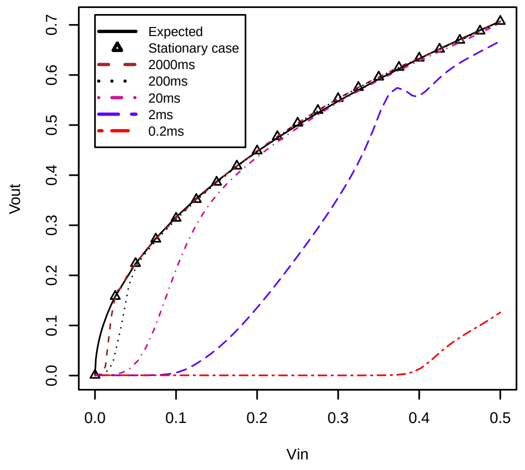
Figure 6: Measured output versus expected output for the best non-computational circuits.



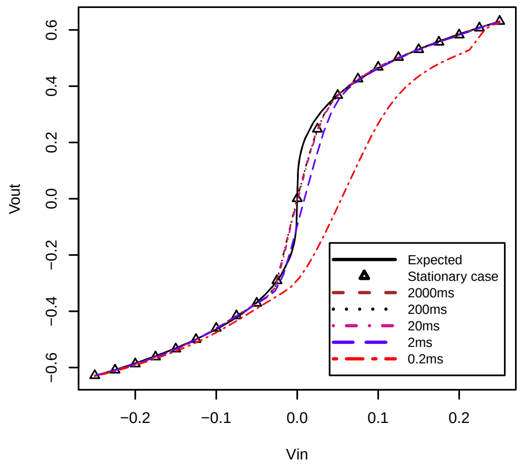
(a) Squaring circuit



(b) Cubing circuit



(c) Square root circuit



(d) Cube root circuit

Figure 7: Measured output versus expected output in the best computational circuits. These results are obtained for a DC sweep analysis in stationary conditions and transient analyses in non-stationary conditions (using different rise times for a ramp input signal). The circuits were evolved using a fitness function based on a DC sweep analysis.

Table 5: Comparison between the one-point and one-block crossover operators in relation to success rate (SR) in the three non-computational circuits. The p-values for one-tailed hypothesis test for difference of proportions are also shown

Goal	MNN	Operator	Runs	SR (%)	p-value
Temp. sensing	6	One-point	50	28.0	0.071
Temp. sensing	6	One-block	50	42.0	
Volt. Ref.	10	One-point	50	14.0	0.067
Volt. Ref.	10	One-block	50	26.0	
Gaussian	15	One-point	50	22.0	0.320
Gaussian	15	One-block	50	26.0	

the one-point operator. In order to evaluate these results from a statistical point of view, a one-tailed hypothesis test for difference of proportions was made. The p-values obtained (see table 5) reveal that the results are statistically significant ($\alpha = 0.1$) for the temperature sensing and voltage reference circuit. Therefore, we can conclude that, for the cases studied, the one-block operator outperforms or equals the one-point operator's results.

5.2. Set of computational circuits

The case studies for the set of computational circuits only focus on the design, so the SR and MBF are not shown here. The use of $3,000$ generations was enough to obtain good results. The MNN values used for each circuit, following the strategy indicated in Section 5.1, were the following: 10 for squaring and square root, 20 for cubing and 30 for cube root. Fig. 7 shows a comparison between the expected and measured output values for the best circuits obtained in stationary conditions.

On the other hand, it is also interesting to analyze the time-domain response of the computational circuits obtained (non-stationary conditions). To do that, a ramp with different rise times, Δ_{rt}, is applied to the input of the four best computational circuits. Fig. 7 also shows the output obtained for each case. All the circuits work fine for a rise time of $\Delta_{rt} = 200ms$ but the output worsens as the value of Δ_{rt} decreases. At this point, it should be recalled that the circuits were evolved from a fitness function based on a DC sweep analysis. However there is evidence in the literature that the evaluation of the fitness function based on this type of analysis can lead to less robust circuit designs [42, 24]. For this reason, a new experiment was run in order to analyze whether it is possible to improve the time-domain response of this kind of circuits. In particular, this study was done by choosing arbitrarily the squaring circuit. Now, the fitness function was based on a time-domain analysis (instead of a DC sweep analysis), where the rise time of the ramp was set to $\Delta_{rt} = 0.2s$. Fig. 8 shows a comparison between the measured and expected output values for the best circuit obtained with this experiment and using ramp input signals with different rise times. As it can be seen in the above-mentioned figure, the behavior of this circuit is much better than that obtained from a fitness function based on a DC sweep analysis (see Fig. 7a). Besides, the expected output is very close to the measured output, even for $\Delta_{rt} = 0.2ms$. From now on, this circuit will be considered the best version of the squaring circuit. The results of this study are consistent with those obtained in [24, 42], where evaluation of the fitness function based on a time-domain analysis appears to be more robust than that based on a DC sweep analysis.

Finally, the topology and sizing of the best computational circuits obtained are included in the final appendix (see Fig. A.12, A.13, A.14 and A.15).

6. Discussion

This section compares the results obtained with our best-evolved circuits with previous works for the two sets of benchmark circuits. In relation to the three non-computational circuits, they

Table 6: Comparison with previous works for the best non-computational circuits. Note that, in AGE [21], the results for #Components are averaged for five circuits, so the value shown is not an integer

	GP [18]	AGE [21]	Proposed work - GE
Temperature sensor			
Fitness	26.4	1.13	**0.065**
#Evaluations	1.60×10^7	6.50×10^6	$\mathbf{6.14 \times 10^6}$
#Components	67	70.2	**33**
Voltage reference			
Fitness	6.6	2.64	**0.112**
#Evaluations	5.12×10^7	5.60×10^6	$\mathbf{1.86 \times 10^6}$
#Components	54	**27.8**	32
Gaussian function generator			
Fitness	0.094	0.3	**0.036**
#Evaluations	2.30×10^7	$\mathbf{4.30 \times 10^6}$	6.23×10^6
#Components	**14**	36	28

were synthesized in previous works using genetic programming [18] and analog genetic encoding [21]. Table 6 compares our results with the mentioned works. The results of previous works are shown as presented in the papers where the methods are published. In order to make a fair comparison, all the fitness values shown are computed with the same fitness function. The specifications of each circuit are the same as Koza's [18] except in the case of the voltage reference circuit, where a different load resistor ($10K\Omega$) is used as it was also done in [21].

As it can be seen in the above-mentioned table, in all the cases, our method improves the fitness results and uses a number of evaluations similar or lower. In addition, the number of components used is the lowest, for the case of the temperature sensor, and is in the middle, for the case of the voltage reference or the Gaussian function generator.

In relation to the four computational circuits, they were synthesized in previous works using genetic programming (GP) [18, 42, 43] and evolution strategies (ES) [24]. The cubing circuit is also compared with a conventional design [44]. Table 7 show our results compared with the mentioned works. The results of the different approaches, with which we compare our results, are shown as presented in [24]. In order to make a fair comparison, which was independent of the fitness function used in each paper, the mean absolute error (MAE) was used, defined as it is indicated in (11), where O_i is the expected output, $\widetilde{O}_i$ is the measured output, n is the number of fitting points, and $i = 1, ..., n$. As it can be seen in the above-mentioned table, our method obtains the minimum MAE with a minimum number of evaluations for the four circuits considered. The number of components used was also competitive in relation to the results obtained with other approaches.

$$MAE = \frac{1}{n} \sum_{i=1}^{i=n} |O_i - \widetilde{O}_i| \tag{11}$$

In addition to the aforementioned advantages of GE over GP (see section 3.1), the use of BNF grammars may also help to explain the good results obtained by our method. In particular, our grammar helps to reduce the search space of the solution in two forms. On the one hand, it allows us to discretize that space in a custom way. For example, the search of capacitor or resistor values is limited to a set of discrete values according to the following format: "$X.YeZ$" (see table 1). On the other hand, it allows us to formalize different types of constraints with the additional advantage that any individual decoded using such grammar will always meet the constraints. For example, the maximum node number (MNN) can be constrained by the

Table 7: Comparison with previous works for the set of computational circuits.

	GP [18]	GP [42]	GP [43]	Manual design[44]	ES [24]	Proposed work - GE
Square root						
Mean absolute error, mV	183.57	20	-	-	9.23	**2.048**
Component no.	64	39	-	-	**22**	26
Evaluation no.	-	6.70×10^9	-	-	3.70×10^6	$\mathbf{1.83 \times 10^6}$
Squaring						
Average error, mV	-	27	-	-	1.44	**0.109**
Component no.	39	37	-	-	35	**29**
Evaluation no.	-	1.00×10^9	-	-	2.70×10^6	$\mathbf{1.87 \times 10^6}$
Cube root						
Average error, mV	80.00	-	-	-	11.90	**4.178**
Component no.	50	-	-	-	39	**28**
Evaluation no.	3.80×10^7	-	-	-	4.50×10^6	$\mathbf{1.85 \times 10^6}$
Cubing						
Average error, mV	1.04	-	0.99	7.13	0.29	**0.0585**
Component no.	56	-	47	**12**	44	36
Evaluation no.	-	-	2.94×10^6	-	2.34×10^6	$\mathbf{1.87 \times 10^6}$

grammar in order to limit the number of nodes that can be used in the evolved circuit (see table 1).

A more detailed interpretation of the topology and sizing for each one of the circuits obtained could be a long and complex issue. In this context, it is necessary to emphasize the open-ended nature associated to the designs obtained by using evolutionary algorithms. This kind of algorithms can connect devices in new and arbitrary ways. On the other hand, the human designer is biased by his domain knowledge and previous experience on similar designs or other sub-circuits that can be combined to produce new circuits. It is therefore considered by many to be an art rather than a science. That kind of bias does not exist in an evolutionary algorithm because it is only guided by optimization of the fitness function which represents the design specifications to be met by the solution circuit. In this sense, an automatically designed circuit could look strange to a human designer [20], but despite this fact, the simulation reveals that the evolved circuit works. A future line of study, which is beyond the scope of this work, could be to analyze the circuits obtained and compare them with other hand-designed circuits, assuming that the same type of problem is solved in both cases.

Finally, in order to contextualize the results obtained in our paper, it is necessary to say that we are obtaining netlists which describe synthesized circuits in terms of topology and component sizing. However, from an industrial point of view, this is only one step in a much broader flow [5]. Specifically, after the netlist is generated, it is converted into a layout. Then this layout is used for creation of process masks in order to conveniently dope a specific area of a silicon wafer (chip). By last, the chip is packaged and tested. If a problem is detected after any of these steps, backtracking to the previous step is needed. In the worst case, redesign of the netlist may be needed.

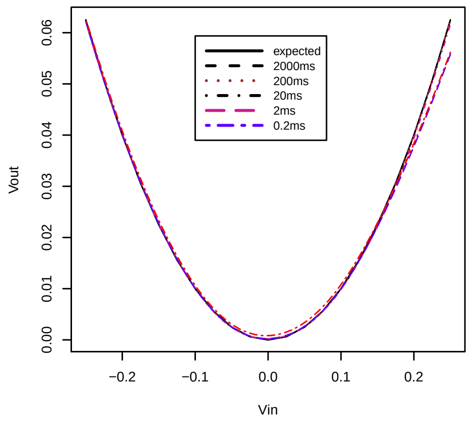


Figure 8: Measured output versus expected output for different rise times of a ramp input signal in the best evolved squaring circuit obtained using a fitness function based on a time-domain analysis.

7. Conclusions

In this paper, we describe the application of GE to the automatic design of analog circuits. A novel grammar is defined in order to generate circuit netlists. The main characteristic of this grammar is that it is oriented to generation of circuit netlists, unlike other approaches based on grammar that are oriented to the building of sub-circuit development. Although GE has been already used to design of digital circuits, this is the first work, as far as authors know, where GE has been applied to design of analog circuits.

Although the maximum node number (MNN) has to be entered by the user before execution, the method is not highly sensitive to the MNN parameter, and a non-zero success rate is obtained for the entire range of MNN values analyzed. In any case, a strategy based on the gradual increase in the MNN was proposed.

Our approach has proved to be a valid and simple method for evolving analog circuits. It was tested in seven benchmark circuits for different purposes: temperature sensor, voltage reference, Gaussian generator function and four computational circuits, each of them implementing a different mathematical function (square root, squaring, cube root and cubing). The circuits obtained were compared with other competitive evolutionary approaches. In relation to the output desired, we obtain the best fitness results with a minimum number of evaluations in the seven cases. Regarding the complexity of the circuits, the method provides circuits with a smaller number of components for 4 out of the 7 types of circuits analyzed.

Finally, for circuits whose output can be sensitive to the frequency of the input signal, we provide evidence that it is more robust to evaluate the fitness function using a time-domain analysis than using a DC sweep analysis.

AppendixA. The best circuits

In this appendix, the topology and sizing of the best circuits obtained with our approach are shown in Fig. A.9-A.15. In order to simplify the figures, resistor blocks comprising resistors in series or parallel were replaced by an equivalent resistor. Additionally, table A.8 shows the results of MOSFET transistor types and channel widths obtained for the Gaussian function generator circuit (see Fig. A.11).

Table A.8: The channel widths and the type of MOSFET transistors, which were obtained during the evolutionary process, for the circuit shown in Fig. A.11

Transistor	Type	Channel width
M1	n-channel	175
M2	n-channel	199
M3	n-channel	74
M4	n-channel	199
M5	p-channel	142
M6	n-channel	11
M7	p-channel	10
M8	n-channel	91
M9	n-channel	199
M10	n-channel	199
M11	n-channel	199
M12	n-channel	99
M13	n-channel	35
M14	n-channel	179
M15	p-channel	46
M16	n-channel	197
M17	p-channel	124

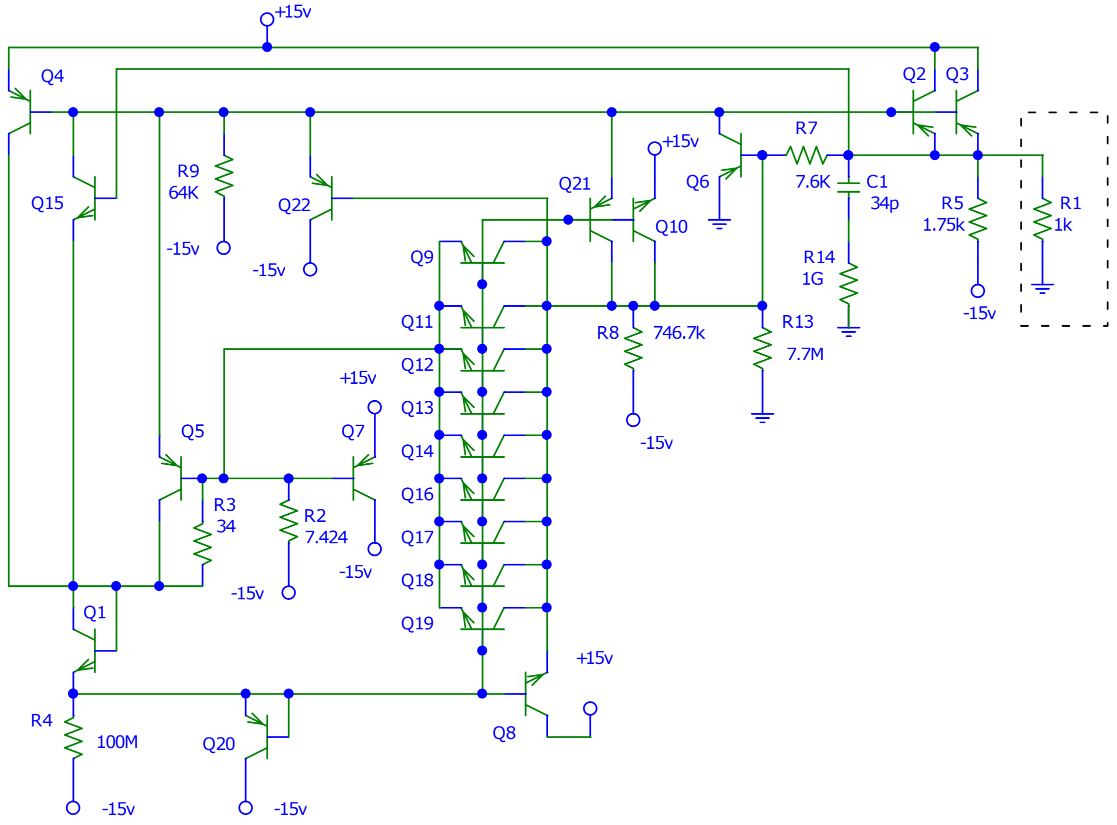


Figure A.9: The best temperature sensing circuit. The dashed line box represents the circuit output. This circuit was evolved from a fitness function based on a temperature sweep analysis.

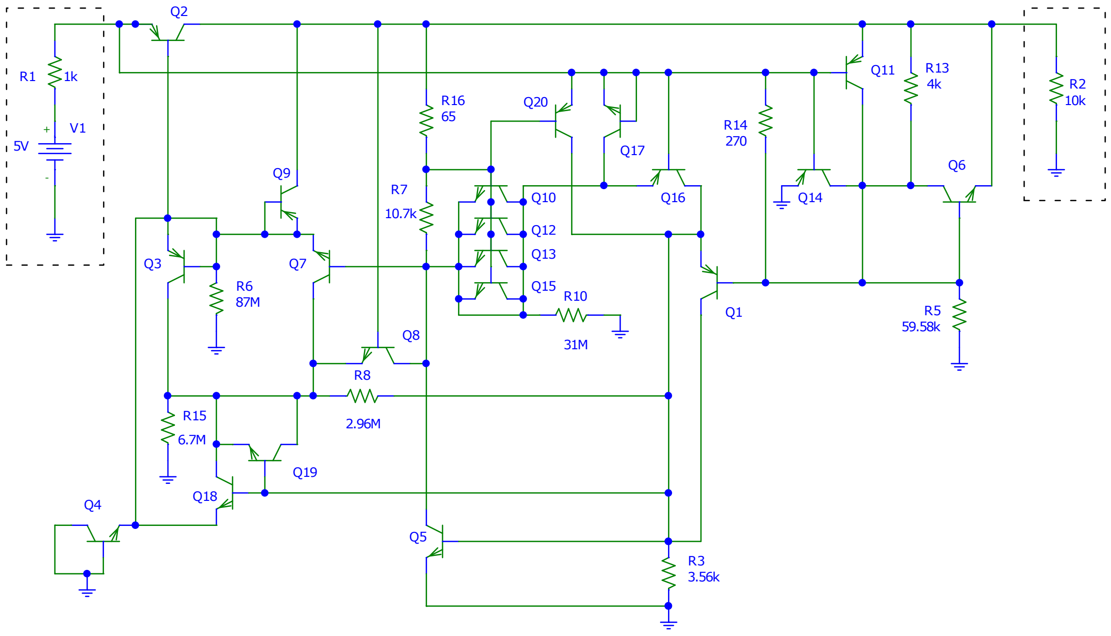


Figure A.10: The best voltage reference circuit. The dashed line boxes represent the input and output, respectively. This circuit was evolved from a fitness function based on a DC sweep analysis.

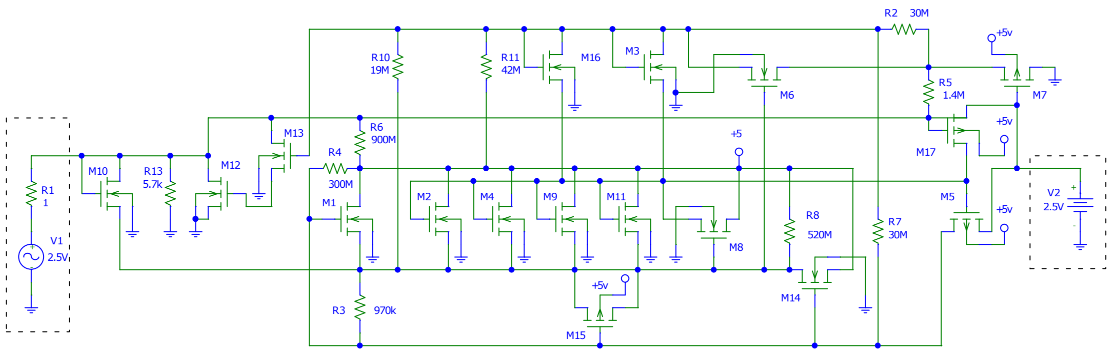


Figure A.11: The best Gaussian function generator circuit. The dashed line boxes represent the input and output, respectively. This circuit was evolved from a fitness function based on a DC sweep analysis.

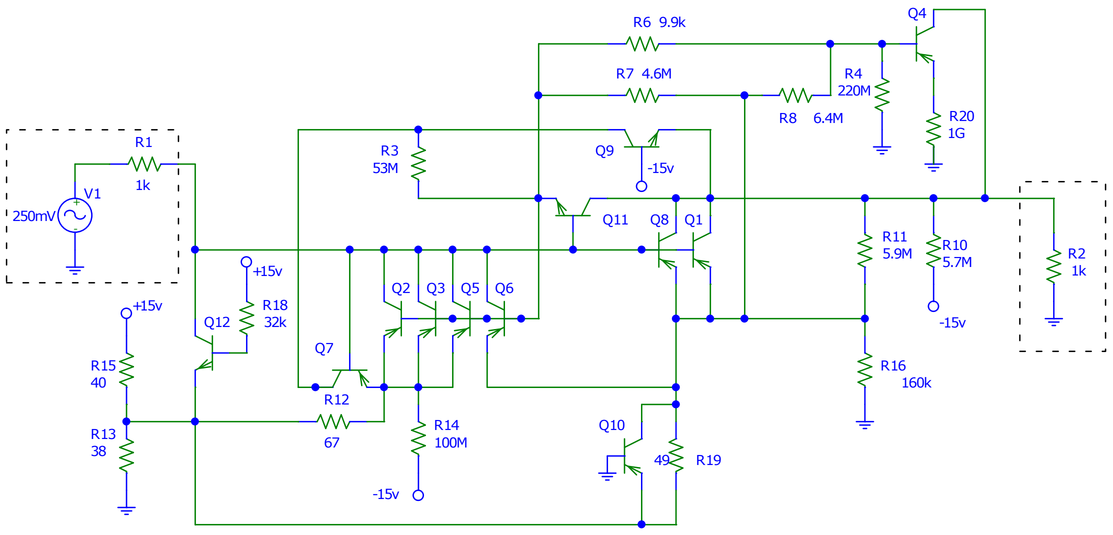


Figure A.12: The best squaring circuit. The dashed line boxes represent the input and output, respectively. This circuit was evolved from a fitness function based on a domain-time analysis.

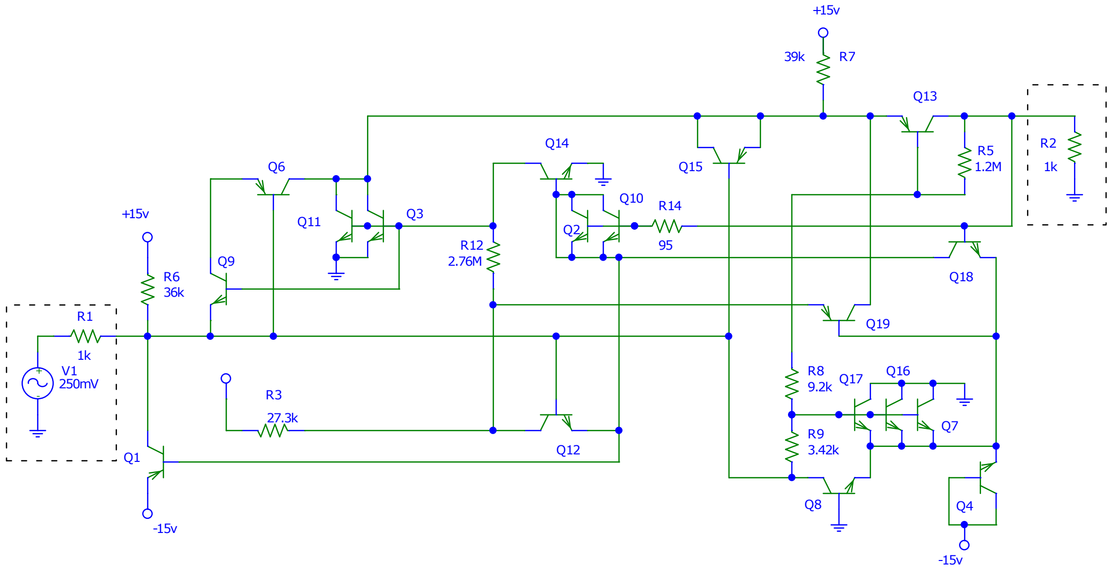


Figure A.13: The best square root circuit. The dashed line boxes represent the input and output, respectively. This circuit was evolved from a fitness function based on a DC sweep analysis.

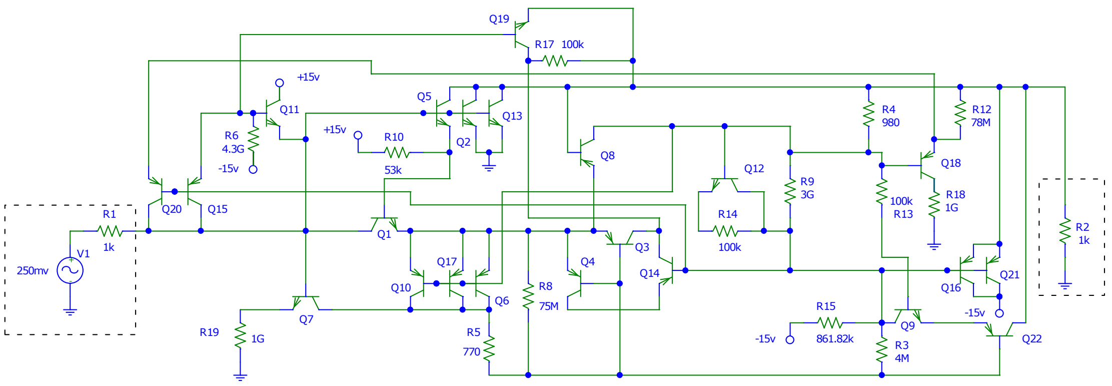


Figure A.14: The best cubing circuit. The dashed line boxes represent the input and output, respectively. This circuit was evolved from a fitness function based on a DC sweep analysis.

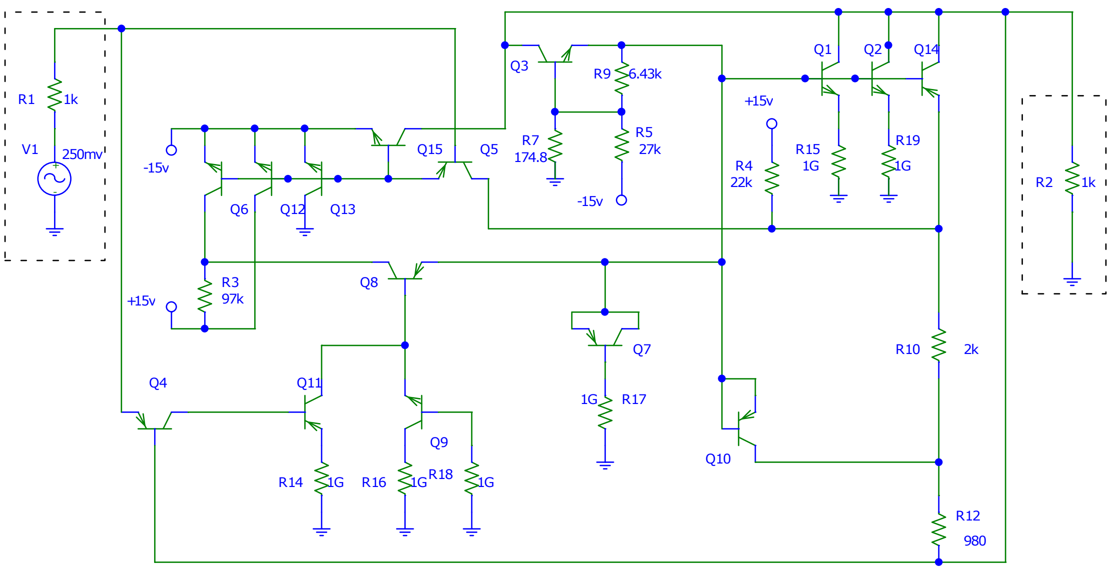


Figure A.15: The best cube root circuit. The dashed line boxes represent the input and output, respectively. This circuit was evolved from a fitness function based on a DC sweep analysis.

References

[1] G. G. E. Gielen, R. A. Rutenbar, Computer-aided design of analog and mixed-signal integrated circuits, in: R. A. Rutenbar, G. G. E. Gielen, B. A. Antao (Eds.), Computer-Aided Design of Analog Integrated Circuits and Systems, John Wiley & Sons, Inc., New York, NY, USA, 2002, pp. 3–10. doi:10.1109/5.899053.

[2] R. Zebulum, M. Pacheco, M. Vellasco, Comparison of different evolutionary methodologies applied to electronic filter design, in: IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence, 1998, pp. 434–439. doi:10.1109/ICEC.1998.699812.

[3] J. Grimbleby, Automatic analogue circuit synthesis using genetic algorithms, IEEE Proceedings - Circuits, Devices and Systems 147 (6) (2000) 319–323. doi:10.1049/ip-cds:20000770.

[4] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing, Natural Computing Series, Springer, 2003.

[5] T. McConaghy, G. Gielen, Genetic programming in industrial analog CAD: Applications and challenges, in: T. Yu, R. Riolo, B. Worzel (Eds.), Genetic Programming Theory and Practice III, Springer US, 2006, pp. 291–306. doi:10.1007/0-387-28111-8_19.

[6] E. Martens, G. Gielen, Classification of analog synthesis tools based on their architecture selection mechanisms, Integration, the VLSI Journal 41 (2) (2008) 238–252. doi:10.1016/j.vlsi.2007.06.001.

[7] D. Nam, Y. D. Seo, L.-J. Park, C. H. Park, B. Kim, Parameter optimization of an on-chip voltage reference circuit using evolutionary programming, IEEE Transactions on Evolutionary Computation 5 (4) (2001) 414–421. doi:10.1109/4235.942535.

[8] J. Ramos, K. Francken, G. G. E. Gielen, M. S. J. Steyaert, An efficient, fully parasitic-aware power amplifier design optimization tool, IEEE Transactions on Circuits and Systems I 52 (8) (2005) 1526–1534. doi:10.1109/TCSI.2005.851677.

[9] C. A. Coello Coello, A. H. Aguirre, Design of combinational logic circuits through an evolutionary multiobjective optimization approach, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 16 (1) (2002) 39–53. doi:10.1017/S0890060401020054.

[10] U. R. Karpuzcu, Automatic Verilog code generation through grammatical evolution, in: Proceedings of the 2005 workshops on Genetic and evolutionary computation, ACM, 2005, pp. 394–397. doi:10.1145/1102256.1102346.

[11] X. Yan, W. Wei, R. Liu, S. Zeng, L. Kang, Designing electronic circuits by means of gene expression programming, in: First NASA/ESA Conference on Adaptive Hardware and Systems, 2006, pp. 194–199. doi:10.1109/AHS.2006.31.

[12] X. Yan, J. Jin, Electronic circuits automatic design algorithm, in: Sixth International Conference on Natural Computation (ICNC), Vol. 5, 2010, pp. 2334–2337. doi:10.1109/ICNC.2010.5584122.

[13] M. Anjomshoa, A. Mahani, S. Sadeghifard, A new automated design and optimization method of CMOS logic circuits based on modified imperialistic competitive algorithm, Applied Soft Computing 21 (2014) 423 – 432. doi:10.1016/j.asoc.2014.04.011.

[14] E. Tlelo-Cuautle, M. Duarte-Villaseñor, Evolutionary electronics: Automatic synthesis of analog circuits by GAs, in: A. Yang, Y. Shan, L. Bui (Eds.), Success in Evolutionary Computation, Vol. 92, Springer Berlin / Heidelberg, 2008, pp. 165–187. doi:10.1007/978-3-540-76286-7_8.

[15] E. Tlelo-Cuautle, I. Guerra-Gómez, M. Duarte-Villaseñor, L. de La Fraga, G. Flores-Becerra, G. Reyes-Salgado, C. Reyes-García, G. Rodriguez-Gómez, Applications of evolutionary algorithms in the design automation of analog integrated circuits, Journal of Applied Sciences 10 (2010) 1859–1872. doi:10.3923/jas.2010.1859.1872.

[16] K.-J. Kim, S.-B. Cho, Automated synthesis of multiple analog circuits using evolutionary computation for redundancy-based fault-tolerance, Applied Soft Computing 12 (4) (2012) 1309 – 1321. doi:10.1016/j.asoc.2011.12.002.

[17] H. Mühlenbein, L. Zinchenko, V. Kureichik, T. Mahnig, Effective mutation rate for probabilistic evolutionary design of analogue electrical circuits, Applied Soft Computing 7 (3) (2007) 1012–1018. doi:10.1016/j.asoc.2006.07.001.

[18] J. Koza, D. Andre, F. Bennett III, M. Keane, Genetic Programming III: Darwinian Invention & Problem Solving, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[19] M. O'Neill, A. Brabazon, Recent patents on genetic programming, Recent Patents on Computer Science 2 (1) (2009) 43–49.

[20] T. McConaghy, P. Palmers, M. Steyaert, G. G. Gielen, Variation-aware structural synthesis of analog circuits via hierarchical building blocks and structural homotopy, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28 (9) (2009) 1281–1294. doi:10.1109/TCAD.2009.2023195.

[21] C. Mattiussi, D. Floreano, Analog genetic encoding for the evolution of circuits and networks, IEEE Transactions on Evolutionary Computation 11 (5) (2007) 596–607. doi:10.1109/TEVC.2006.886801.

[22] R. S. Zebulum, M. A. Pacheco, M. Vellasco, Artificial evolution of active filters: A case study, in: Proceedings of the 1st NASA/DOD Workshop on Evolvable Hardware, IEEE Computer Society, 1999, pp. 66–75. doi:10.1109/EH.1999.785436.

[23] R. S. Zebulum, M. S. Vellasco, M. A. Pacheco, Variable length representation in evolutionary electronics, Evolutionary Computation 8 (1) (2000) 93–120. doi:10.1162/106365600568112.

[24] Y. A. Sapargaliyev, T. G. Kalganova, Open-ended evolution to discover analogue circuits for beyond conventional applications, Genetic Programming and Evolvable Machines 13 (4) (2012) 411–443. doi:10.1007/s10710-012-9163-8.

[25] S. Ando, H. Iba, Analog circuit design with a variable length chromosome, in: Proceedings of the 2000 Congress on Evolutionary Computation, Vol. 2, 2000, pp. 994–1001. doi:10.1109/CEC.2000.870754.

[26] J. B. Grimbleby, Automatic analogue network synthesis using genetic algorithms, in: First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995, pp. 53–58.

[27] J. Koza, F. Bennett III, D. Andre, M. Keane, Synthesis of topology and sizing of analog electrical circuits by means of genetic programming, Computer Methods in Applied Mechanics and Engineering 186 (2-4) (2000) 459–482. doi:10.1016/S0045-7825(99)00397-7.

[28] J. Koza, F. Bennett III, D. Andre, M. Keane, Evolutionary design of analog electrical circuits using genetic programming, in: I. C. Parmee (Ed.), Adaptive Computing in Design and Manufacture: The Integration of Evolutionary and Adaptive Computing Technologies with Product/System Design and Realisation, Springer London, 1998, pp. 177–192. doi:10.1007/978-1-4471-1589-2_14.

[29] J. Koza, F. Bennett III, D. Andre, M. Keane, The design of analogue circuits by means of genetic programming, in: P. J. Bentley (Ed.), Evolutionary Design by Computers, John Wiley&Son, 1999, Ch. 16, pp. 365–385.

[30] J. Koza, F. Bennett III, D. Andre, M. Keane, Automatic design of analog electrical circuits using genetic programming, in: H. Cartwright (Ed.), Intelligent Data Analysis in Science, Oxford University Press, Oxford, 2000, Ch. 8, pp. 172–200.

[31] F. Castejón, E. Carmona, Automatic design of electronic amplifiers using grammatical evolution, in: A. Alonso-Betanzos, et al. (Eds.), Actas de Multiconferencia CAEPIA-13, 2013, pp. 703–712.

[32] C. Mattiussi, Evolutionary synthesis of analog networks, Ph.D. thesis, Università degli Studi di Trieste (2005).

[33] M. O'Neill, C. Ryan, Grammatical evolution, IEEE Transactions on Evolutionary Computation 5 (2001) 349–358. doi:10.1109/4235.942529.

[34] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, M. O'Neill, Grammar-based genetic programming: a survey, Genetic Programming and Evolvable Machines 11 (3-4) (2010) 365–396. doi:10.1007/s10710-010-9109-y.

[35] ISO/IEC-14977, Information technology – syntactic metalanguage – extended BNF (1996).

[36] F. Wang, Y. Li, L. Li, K. Li, Automated analog circuit design using two-layer genetic programming, Applied Mathematics and Computation 185 (2) (2007) 1087–1097. doi:10.1016/j.amc.2006.07.029.

[37] J. D. Lohn, S. P. Colombano, A circuit representation technique for automated circuit design, IEEE Transactions on Evolutionary Computation 3 (3) (1999) 205–219. doi:10.1109/4235.788491.

[38] P. Nenzi, H. Vogt, Ngspice user's manual version 23 (2011).

[39] L. W. Nagel, D. O. Pederson, SPICE: Simulation program with integrated circuit emphasis, Electronics Research Laboratory, College of Engineering, University of California, 1973.

[40] T. McConaghy, G. Gielen, Canonical form functions as a simple means for genetic programming to evolve human-interpretable functions, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM, 2006, pp. 855–862. doi:10.1145/1143997.1144147.

[41] M. Nicolau, I. Dempsey, Introducing grammar based extensions for grammatical evolution, in: IEEE Congress on Evolutionary Computation, IEEE, 2006, pp. 648–655.

[42] W. Mydlowec, J. Koza, Use of time-domain simulations in automatic synthesis of computational circuits using genetic programming, in: Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, 2000, pp. 187–197.

[43] M. J. Streeter, M. A. Keane, J. R. Koza, Iterative refinement of computational circuits using genetic programming, in: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers Inc., 2002, pp. 877–884.

[44] S. Cipriani, A. Takeshian, Compact cubic function generator, uS Patent 6,160,427 (Dec. 12 2000).

[45] M. O'Neill, C. Ryan, Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language, Kluwer Academic Publishers, 2003. doi:10.1007/978-1-4615-0447-4.