AUTOMATED DESIGN OF BOTH THE TOPOLOGY AND SIZING OF ANALOG ELECTRICAL CIRCUITS USING GENETIC PROGRAMMING

JOHN R. KOZA, FORREST H BENNETT III, DAVID ANDRE
Stanford University, Computer Science Department, Stanford, California

AND

MARTIN A. KEANE
Econometrics Inc., Chicago, Illinois USA

Abstract: This paper describes an automated process for designing analog electrical circuits based on the principles of natural selection, sexual recombination, and developmental biology. The design process starts with the random creation of a large population of program trees composed of circuit-constructing functions. Each program tree specifies the steps by which a fully developed circuit is to be progressively developed from a common embryonic circuit appropriate for the type of circuit that the user wishes to design. Each fully developed circuit is translated into a netlist, simulated using a modified version of SPICE, and evaluated as to how well it satisfies the user's design requirements. The fitness measure is a user-written computer program that may incorporate any calculable characteristic or combination of characteristics of the circuit, including the circuit's behavior in the time domain, its behavior in the frequency domain, its power consumption, the number of components, cost of components, or surface area occupied by its components. The population of program trees is genetically bred over a series of many generations using genetic programming. Genetic programming is driven by a fitness measure and employs genetic operations such as Darwinian reproduction, sexual recombination (crossover), and occasional mutation to create offspring. This automated evolutionary process produces both the topology of the circuit and the numerical values for each component. This paper describes how genetic programming can evolve the circuit for a difficult-to-design low-pass filter.

1. The Problem of Circuit Design

The design of an electrical circuit with specified operating characteristics is a complex task. Electrical circuits consist of a wide variety of different types of components, including wires, resistors, capacitors, inductors, diodes, transistors, transformers, and energy sources. The individual components are arranged in a particular *topology* to form a closed circuit. In addition, each component is further specified (*sized*) by a set of component values. Circuits typically receive input signals from one or more input sources and produce output signals at one or more output ports. A complete specification of an electrical circuit includes both its topology and the sizing of all of its components.

Considerable progress has been made in automating the design of certain categories of purely digital circuits; however, the design of analog circuits and mixed analog-

digital circuits has not proved to be as amenable to automation (Rutenbar 1993). In discussing "the analog dilemma," Aaserud and Nielsen (1995) observe,

> "Analog designers are few and far between. In contrast to digital design, most of the analog circuits are still handcrafted by the experts or so-called 'zahs' of analog design. The design process is characterized by a combination of experience and intuition and requires a thorough knowledge of the process characteristics and the detailed specifications of the actual product.
>
> "Analog circuit design is known to be a knowledge-intensive, multiphase, iterative task, which usually stretches over a significant period of time and is performed by designers with a large portfolio of skills. It is therefore considered by many to be a form of art rather than a science. "

2. Previous Work

Numerous efforts have been made to automate the design process for analog and mixed analog-digital circuits. In an interactive design tool called IDAC for analog integrated circuits (Degrauwe 1987), the user selects various possible topologies for the circuit; IDAC determines the values of the components in each circuit (in relation to the desired behavioral characteristics); and, the user chooses the best sized circuit.

In OASYS (Harjani, Rutenbar, and Carley 1989) and OPASYN (Koh, Sequin, and Gray 1990), a topology is chosen beforehand based on heuristic rules and the synthesis tool attempts to size the circuit. If the synthesis tool cannot size the chosen topology correctly, the tool creates a new topology using other heuristic rules and the process continues. The success of these systems depends on the effectiveness of the knowledge base of heuristic rules.

In SEAS (Ning, Kole, Mouthaan, and Wallings 1992), evolution is used to modify the topology and simulated annealing is used to size the circuit. Maulik, Carley, and Rutenbar (1992) attempt to handle topology selection and circuit sizing simultaneously using expert design knowledge. Higuchi et al. (1993) have employed genetic methods to the design of digital circuits using a hardware description language (HDL).

In DARWIN (Kruiskamp and Leenaerts 1995), opamp circuits are designed using the genetic algorithm (Holland 1975). In creating the initial population in DARWIN, the topology of each opamp in the population is picked randomly from a preestablished hand-designed set of 24 topologies in order to ensure that each circuit behaves as an opamp. In addition, a set of problem-specific constraints are solved to ensure that all transistors operate in their proper range and that all transistor sizes are between maximal and minimal values. The behavior of each opamp is evaluated using a small signal equivalent circuit and analytical calculations specialized to opamp circuits. The fitness of each opamp is computed using a combination of factors, including the deviation between the actual behavior of the circuit and the desired behavior and the power dissipation of the circuit. A crossover operation and mutation operation for the chromosome strings describing the opamps is used to create offspring chromosomes.

3. Background of Genetic Programming

John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975) described how an analog of the naturally-occurring evolutionary process can be applied to solving scientific and engineering problems using what is now called the *genetic algorithm*

The problem of automatic programming is one of the central questions in computer science. Paraphrasing Arthur Samuel (1959), the question is

> How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what needs to be done, without being told exactly how to do it?

Genetic programming *is* automatic programming.

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992) describes an extension of the genetic algorithm in which the genetic population consists of computer programs (that is, compositions of primitive functions and terminals). Genetic programming starts with a primordial ooze of randomly generated computer programs composed of the available programmatic ingredients and then applies the principles of animal husbandry to breed a new (and often improved) population of programs. The breeding is done in a domain-independent way using the Darwinian principle of survival of the fittest, an analog of the naturally-occurring genetic operation of crossover (sexual recombination), and occasional mutation. The crossover operation is designed to create syntactically valid offspring programs (given closure amongst the set of ingredients). Genetic programming combines the expressive high-level symbolic representations of computer programs with the near-optimal efficiency of learning of Holland's genetic algorithm. A computer program that solves (or approximately solves) a given problem often emerges from this process.

Genetic programming breeds computer programs to solve problems by executing the following three steps:

> (1) Generate an initial population of random compositions of the functions and terminals of the problem (i.e., computer programs).
>
> (2) Iteratively perform the following substeps until the termination criterion has been satisfied:
>
> (A) Execute each program in the population and assign it a fitness value using the fitness measure.
>
> (B) Create a new population of computer programs by applying the following operations. The operations are applied to computer program(s) chosen from the population with a probability based on fitness.
>
> (i) *Darwinian Reproduction*: Reproduce an existing program by copying it into the new population.
>
> (ii) *Crossover*: Create two new computer programs from two existing programs by genetically recombining randomly chosen parts of two existing programs using the crossover operation (described below) applied at a randomly chosen crossover point within each program.
>
> (iii) *Mutation*: Create one new computer program from one existing program by mutating a randomly chosen part of the program.

(3) The program that is identified by the method of result designation (e.g., the best-so-far individual) is designated as the result of the genetic algorithm for the run. This result may be a solution (or an approximate solution) to the problem.

The genetic crossover operation operates on two parental computer programs selected with a probability based on fitness and produces two new offspring programs consisting of parts of each parent. Because entire sub-trees are swapped, the crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points (assuming closure among the functions and terminals involved). Because programs are selected to participate in the crossover operation with a probability based on fitness, crossover allocates future trials to regions of the search space whose programs contain parts from promising programs.

The genetic mutation operation operates on one parental computer program selected with a probability based on fitness and produces one new offspring program. A point is randomly chosen in the parental program; the subtree rooted at that point is deleted from the program; and a new subtree is randomly grown at that point using the available functions and terminals (usually in the same manner as trees are grown in creating the initial random population). Mutation is used sparingly in genetic programming.

Genetic Programming II: Automatic Discovery of Reusable Programs (Koza 1994) demonstrates that genetic programming can evolve multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms (called *automatically defined functions* or *ADF*s).

4. Background on Cellular Encoding of Neural Networks

A feedforward neural network is a complex structure that can be represented by line-labeled, point-labeled, directed graph. The points of the graph are either neural processing units within the network, input points, or output points. The lines are labeled with weights to represent the weighted connections between two points. The neural processing units are labeled with numbers indicating both the threshold and the bias of the processing unit.

In his seminal *Cellular Encoding of Genetic Neural Networks*, Frederic Gruau (1992) described an innovative and clever technique, called *cellular encoding*, in which genetic programming is used to concurrently evolve the architecture of a neural network, along with all weights, thresholds, and biases. In cellular encoding, each individual program tree in the population is a specification for developing a complete neural network from a very simple embryonic neural network (consisting of a single neuron). Genetic programming is applied to populations of these network-constructing program trees in order to evolve a neural network capable of solving the problem at hand. See also Gruau 1994.

Each program tree is a composition of network-constructing, neuron-creating, and neuron-adjusting functions and terminals. The program tree is the genotype and the neural network constructed in accordance with the tree's instructions is the phenotype. The fitness of an individual program tree in the population is measured by how well the neural network that is constructed in accordance with the instructions contained in the program tree performs the desired task. Genetic programming then breeds the population of program trees in the usual manner using Darwinian reproduction, crossover, and mutation.

5. Background on SPICE

SPICE (an acronym for Simulation Program with Integrated Circuit Emphasis) is a massive family of programs written over several decades at the University of California at Berkeley for the simulation of analog, digital, and mixed analog/digital electrical circuits (Quarles et al. 1994). The input to a SPICE simulation consists of a netlist describing the circuit to be analyzed and certain commands that instruct SPICE as to the type of analysis to be performed and the nature of the output to be produced.

6. The Mapping between Circuits and Program Trees

Genetic programming breeds a population of rooted, point-labeled trees (i.e., graphs without cycles) with ordered branches.

There is a considerable difference between the kind of trees bred in the world of genetic programming and the special kind of labeled graphs employed in the world of circuits.

Genetic programming can be applied to circuits if a mapping is established between the kind of point-labeled trees found in the world of genetic programming and the line-labeled (often doubly labeled) cyclic graphs employed in the world of circuits. In our case, developmental biology provides the motivation for this mapping. The growth process used herein begins with a very simple embryonic electrical circuit. The circuit is developed as the functions in the program tree are progressively executed. The result is both the topology of the circuit and the sizing of all of its components.

Each program tree contains (1) circuit-constructing functions and terminals that create the topology of circuit from the embryonic circuit, (2) component-setting functions that convert wires (and other components) within the circuit into specified components, and (3) arithmetic-performing functions and numerical terminals that together specify the numerical value (sizing) for each component of the circuit.

Program trees conform to a constrained syntactic structure. Component-setting functions have arithmetic-performing argument subtrees and construction-continuing argument subtrees, while the circuit-constructing functions that manipulate the topology of the circuit have one or more construction-continuing argument subtrees. The left argument subtree of each component-setting function consists of a composition of arithmetic functions and numerical constant terminals that together yield the numerical value for the component. The right argument subtree of each component-setting function specifies how the construction of the circuit is to be continued. Both the random program trees in the initial population (generation 0) and any random subtrees created by the mutation operation in later generations are created so as to conform to this constrained syntactic structure. This constrained syntactic structure is preserved by the crossover operation using structure-preserving crossover with point typing.

7. The Embryonic Circuit

An electrical circuit is created by executing the program tree. Each program tree in the population creates one electrical circuit from the common embryonic circuit.

The embryonic circuit used on a particular problem depends on the number of input signals and the number of output signals (probe points). It may also contain certain fixed components that are required or desired for the circuit being designed.

The embryonic circuit used herein contains one input signal, one probe point, two modifiable wires, a fixed source resistor, and a fixed load resistor. In the embryonic circuit, the two modifiable wires each initially possess a writing head (i.e., are highlighted with a circle). A circuit is developed by modifying the component to which a writing head is pointing in accordance with the circuit-constructing functions in the program tree. Each circuit-constructing function in the program tree changes its associated highlighted component in the developing circuit in a particular way and specifies the future disposition of successor writing head(s), if any.

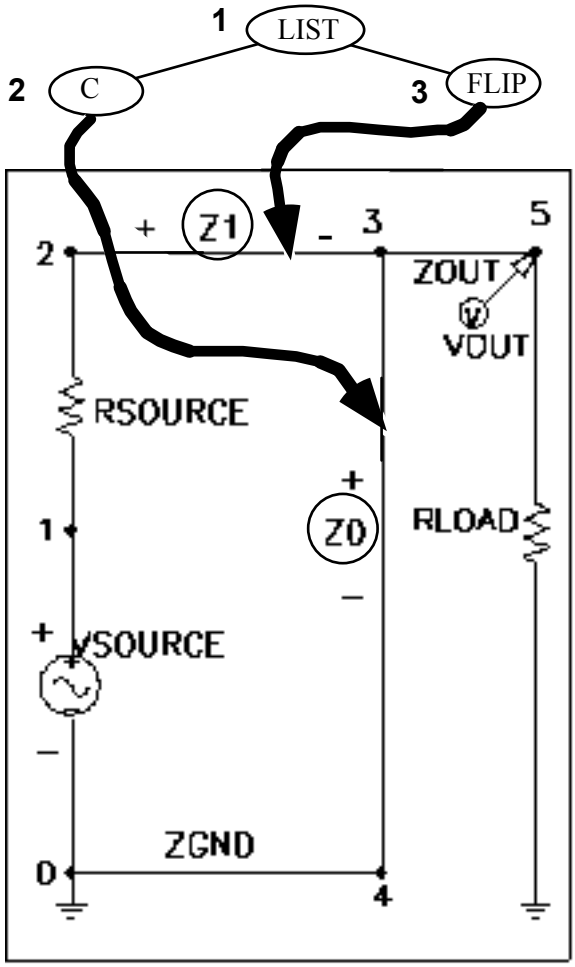

Figure 1 One-input, one-output embryonic electrical circuit.

The bottom three quarters of figure 1 shows the embryonic circuit used for a one-input, one-output circuit. The energy source is a 2 volt sinusoidal voltage source VSOURCE whose negative (−) end is connected to node 0 (ground) and whose positive (+) end is connected to node 1. There is a fixed 1000-Ohm source resistor RSOURCE between nodes 1 and 2. There is a modifiable wire (i.e., a wire with a writing head) Z1

between nodes 2 and 3 and another modifiable wire Z0 between nodes 3 and 4. There are circles around modifiable wires Z0 and Z1 to indicate that the two writing heads (thick lines) point to them. There is a fixed isolating wire ZOUT between nodes 3 and 5, a voltage probe labeled VOUT at node 5, and a fixed 1000-Ohm load resistor RLOAD between nodes 5 and 0 (ground). There is an isolating wire ZGND between nodes 4 and 0 (ground). All of the above elements of this embryonic circuit (except Z0 and Z1) are fixed forever; they are not subject to modification during the process of developing the circuit. All subsequent development of the circuit originates from writing heads.

A circuit is developed by modifying the component to which a writing head is pointing in accordance with the associated circuit-constructing function in the program tree. The figure shows L and FLIP functions just below the LIST and the two writing heads pointing to modifiable wires Z0 and Z1. The L and FLIP functions will cause Z0 to be changed into a capacitor and the polarity of modifiable wire Z1 to be reversed.

The embryonic circuit is designed so that the number of lines impinging at any one node in the circuit is either two or three. This condition is maintained by all of the circuit-constructing functions. The isolating wire ZOUT protects the probe point VOUT from modification during the developmental process and the isolating wire ZGND protects the negative terminal of VSOURCE.

Note that little domain knowledge went into this embryonic circuit. Specifically, (1) the embryonic circuit is a circuit, (2) the embryonic circuit has one input and one output, and (3) there are modifable connections between the output and both source and ground. This embryonic circuit is applicable to any one-input, one-output circuit. It is the fitness measure that directs the evolutionary search process to the desired circuit.

8. Circuit-Constructing Functions

Each circuit-constructing function operates on a single component.

8.1. THE C AND L COMPONENT-SETTING FUNCTIONS

Components are introduced into a circuit by the component-setting functions.

The rightmost argument subtree of each component-setting function is a construction-continuing subtree that points to a successor function or terminal in the program tree. Upon completion, one writing head points to the new component.

The left argument subtree of the component-setting functions is an arithmetic-performing subtree that contains a composition of arithmetic functions (addition and subtraction) and random constants (in the range –1.000 to +1.000). The arithmetic-performing subtree returns a floating-point value which is, in turn, interpreted as the value of the component using a logarithmic scale in the following way: If the return value is between –5.0 and +5.0, U is equated to the value returned by the argument subtree . If the return value is less than –100 or greater than +100, U is set to zero. If the return value is between –100.0 and –5.0, U is found from the straight line connecting the points (–100,0) and (–5, -5). If the return value is between +5.0 and +100, U is found from the straight line connecting (5,5) and (100, 0). The value of the component is 10^U in a unit that is appropriate for the type of component. This mapping gives the component a value within a range of 11 orders of magnitude centered on a certain value.

This mapping gives the component a value within a range of 11 orders of magnitude that is centered on an appropriate value and that uses an appropriate unit of measurement that was settled upon after examining a large number of practical circuits in contemporary books.

If a component (e.g., a diode) has no numerical values, there is no left argument subtree.

The two-argument C ("capacitor") function causes the highlighted component to be changed into a capacitor. The value of the capacitor is the antilogarithm (base 10) of the intermediate value *U* computed as above in nano-Farads (nF). This mapping gives the capacitor a value within a range of plus or minus 5 orders of magnitude centered on 1nF.

The two-argument L ("inductor") function causes the highlighted component to be changed into an inductor. The value of the inductor is the antilogarithm (base 10) of the intermediate value U in micro-Henrys (mH).

8.2. THE FLIP FUNCTION

All electrical components in SPICE have a designated positive (+) end and a designated negative (−) end. Polarity clearly matters for components such as diodes and transistors and it affects the course of the developmental process for all components. The one-argument FLIP function attaches the positive end of the highlighted component to the node to which its negative end is currently attached and vice versa. Upon completion, one writing head points to the now-flipped original component.

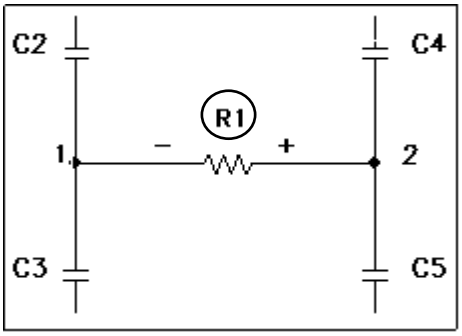

Figure 2 A circuit containing a resistor R1.

8.3. SERIES DIVISION

The three-argument SERIES ("series division") function operates on one highlighted component and creates a series composition consisting of the highlighted component, a copy of the highlighted component, one new modifable wire, and two new nodes. After execution of the SERIES function, there are three writing heads pointing to the original component, the new modifiable wire, and the copy of the original component.

Figure 2 shows a resistor R1 connecting nodes 1 and 2 of a partial circuit containing various capacitors. R1 is assumed to possess a writing head (i.e., is highlighted).

Figure 3 illustrates the result of applying the SERIES division function to resistor R1 from figure 2. First, the SERIES function creates two new nodes, 3 and 4. Second, SERIES relabels the positive (+) end of R1 (currently labeled 2) as the first new node,

3. Third, SERIES creates a new wire Z6 between the first new node, 3, and the second new node, 4. Fourth, SERIES inserts a duplicate (called R7) of the original component (including all its component values) between new node 4 and original node 2.

Note our convention of globally numbering components consecutively (rather than maintaining a different series of consecutive numbers for each type of component). Also, note that wires (such as Z6) are used only during the developmental process; all wires are edited out prior to the final creation of netlist for SPICE. Also, note that the SERIES function may be applied to a wire; in that event, the result is a series composition of three wires (each with its own writing head).

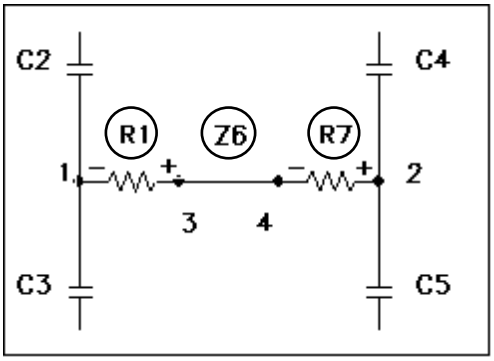

Figure 3 Result after applying the series division function SERIES to resistor R1.

8.4. PARALLEL DIVISION FUNCTIONS

The two four-argument parallel division functions (PSS and PSL) each operate on one highlighted component to create a parallel composition consisting of the original highlighted component, a duplicate of the highlighted component, two new wires, and two new nodes. After execution of a parallel division, there are four writing heads. They point to the original component, the two new modifiable wires, and the copy of the original component. We describe (and use) only PSS herein.

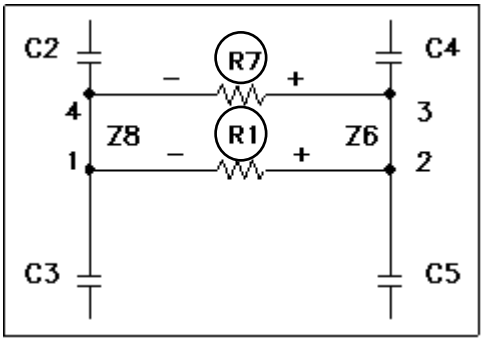

Figure 4 Result after applying PSS to resistor R1.

First, the parallel division function PSS creates two new nodes, 3 and 4. Second, the parallel division function inserts a duplicate of the highlighted component (including all of its component values) between the new nodes 3 and 4 with the negative end of the

duplicate connected to node 4 and the positive end of the duplicate connected to node 3. Third, the parallel division function creates a first new wire Z6 between the positive (+) end of R1 (which is at original node 2) and first new node, 3. Fourth, the parallel division function creates a second new wire Z8 between the negative (-) end of R1 (which is at original node 1) to second new node, 4.

The second character (i.e., the first S or L) of the name of the particular parallel division function indicates whether the positive end of the new component is connected to the smaller (S) or larger (L) numbered component of the two components that were originally connected to the positive end of the highlighted component. The third character (i.e., the second S or L) of the name of the particular parallel division function indicates whether the negative end of the new component is connected to the smaller (S) or larger (L) numbered component of the two components that were originally connected to the negative end of the highlighted component.

Figure 4 shows the results of applying the PSS function to resistor R1 from figure 2. Since C4 bears a smaller number than C5, new node 3 and new wire Z6 are located between original node 2 and C4. Since C2 bears a smaller number than C3, new node 4 and new wire Z8 are located between original node 1 and C2.

8.5. THE VIA AND GND FUNCTIONS

Eight two-argument functions (called VIA0, ..., VIA7) and the two-argument GND ("ground") function enable distant parts of a circuit to be connected together.

The eight two-argument VIA0, ..., VIA7 functions create a series composition consisting of two wires that each possesses a successor writing head and a numbered port (called a *via*) that possesses no writing head. The port is connected to a designated one of eight imaginary layers (numbered from 0 to 7) in the wafer on which the circuit resides. If one or more other parts of the circuit connects to a particular layer, all such parts become electrically connected as if wires were running between them. If no other part of the circuit connects to a particular layer, then the one port connecting to the layer is useless (and this port is deleted when the netlist for the circuit is eventually created).

The two-argument GND ("ground") function is a special "via" function that connects directly to the electrical ground of the circuit. This direct connection to ground is made even if there is only one GND function calling for a connection to ground in the circuit.

After execution of these functions, writing heads point to the two new wires.

8.6. THE NOP FUNCTION

The one-argument NOP function has no effect on the highlighted component; however, it delays activity on the developmental path on which it appears in relation to other developmental paths in the overall program tree – thereby (possibly) affecting the overall result produced by the construction process. After execution of NOP, one writing head points to the original highlighted component.

8.7. THE END FUNCTION

The zero-argument END function causes the highlighted component to lose its writing head – thereby ending that particular developmental path.

9. The Problem of Designing a Lowpass LC Filter

Consider a circuit design problem in which the goal is to design a filter using inductors and capacitors with an AC input signal with 2 volt amplitude. The filter is have a passband below 1,000 Hertz with voltage values between 970 millivolts and 1 volt and to have a stopband above 2,000 Hz with voltage values between 0 volts and 1 millivolts. This corresponds to a pass band ripple of at most 0.3 decibels and a stop band attenuation of at least 60 decibels. The circuit is to be driven from a source with an internal (source) resistance of 1,000 Ohms and terminated in a load of 1,000 Ohms.

A practising engineer would regard finding a circuit satisying the requirements as a non-trivial design problem. Using the terminology of Zverev (1967), these requirements can be satisfied by a Chebyshev-Cauer filter of complexity 5, with a relection coefficient of 20%, and modular angle of 30 degrees.

10. Preparatory Steps for Solving the Problem of Designing a Lowpass LC Filter

Before applying genetic programming to a circuit design problem, the user must perform seven major preparatory steps, namely (1) identifying the terminals of the to-be-evolved programs, (2) identifying the primitive functions contained in the to-be-evolved programs, (3) creating the fitness measure for evaluating how well a given program does at solving the problem at hand, (4) choosing certain control parameters (notably population size and the maximum number of generations to be run), (5) determining the termination criterion and method of result designation (typically the best-so-far individual from the populations produced during the run), (6) determining the architecture of the overall program, and (7) identifying the embryonic circuit that is suitable for the problem.

Since the problem of designing the lowpass LC filter calls for a one-input, one-output circuit with a source resistor and a load resistor, we use the embryonic circuit of figure 2 for this problem. Since the embryonic circuit starts with two writing heads, each program tree has two result-producing branches joined by a `LIST` function. There are no automatically defined functions. The terminal set and function set for both result-producing branches are the same. Each result-producing branch is created in accordance with the constrained syntactic structure that uses the leftmost (first) argument(s) of each component-creating function to specify the numerical value of the component. The numerical value is created by a composition of arithmetic functions and random constants in this arithmetic-performing subtree. Since the components involved in this problem (i.e., inductors and capacitors) each take exactly one component value, there is only one arithmetic-performing subtree. The rightmost (second) argument of each component-creating function is then used to continue the program tree.

In particular, the function set, $\mathcal{F}_{aps}$ for the arithmetic-performing subtree associated with each component-creating function contains the two-argument functions of addition and subtraction. That is,

$$\mathcal{F}_{aps} = \{+, -\}.$$

The terminal set, $\mathcal{T}_{aps}$, for the arithmetic-performing subtree consists of

$$\mathcal{T}_{aps} = \{\leftarrow\},$$

where ← represents floating-point random constants between –1.000 and +1.000.

The function set, $\mathcal{F}_{\mathrm{CCS}}$, for the construction-continuing subtree of each component-creating function is

$$\mathcal{F}_{\mathrm{CCS}} = \{\mathtt{C}, \mathtt{L}, \mathtt{SERIES}, \mathtt{PSS}, \mathtt{FLIP}, \mathtt{NOP}, \mathtt{GND}, \mathtt{VIA0}, \mathtt{VIA1}, \mathtt{VIA2}, \mathtt{VIA3}, \mathtt{VIA4}, \mathtt{VIA5}, \mathtt{VIA6}, \mathtt{VIA7}\},$$

taking 2, 2, 3, 4, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, and 2 arguments, respectively. The terminal set, $\mathcal{T}_{\mathrm{CCS}}$, for the construction-continuing subtree consists of

$$\mathcal{T}_{\mathrm{CCS}} = \{\mathtt{END}\}.$$

The user provides a computer program to compute the fitness measure. The fitness measure drives the evolutionary process. For this problem, the voltage VOUT is probed at node 5 and the circuit is viewed in the frequency domain.

Note that the above is applicable to any one-input, one-output LC circuit. It is the fitness measure that directs the evolutionary process to the desired circuit.

Each circuit that is developed from the embryonic circuit is simulated using a modified version of the 217,000-line SPICE simulator that we modified to run as a submodule of our genetic programming system. The SPICE simulator is requested to perform an AC small signal analysis and to report the circuit's behavior for each of 101 frequency values chosen from the range between 101 frequency values chosen over five decades of frequency (from 1 Hz to 100,000 Hz). Each decade is divided into 20 parts (using a logarithmic scale).

Fitness is measured in terms of the sum, over these 101 fitness cases, of the absolute weighted deviation between the actual value of the voltage in the frequency domain) that is produced by the circuit at the probe point VOUT at node 5 and the target value for voltage. The smaller the value of fitness, the better. A fitness of zero is ideal.

The fitness measure does not penalize ideal values; it slightly penalizes every acceptable deviation; and it heavily penalizes every unacceptable deviation.

The procedure for each of the 61 points in the 3-decade interval from 1 Hz to 1,000 Hz is as follows: If the voltage equals the ideal value of 1.0 volts in this interval, the deviation is 0.0. If the voltage is between 970 millivolts and 1,000 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 1.0. If the voltage is less than 970 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 10.0. This arrangement reflects the fact that the ideal voltage in the passband is 1.0 volt, the fact that a 30 millivolt shortfall is acceptable, and the fact that a voltage below 970 millivolts in the passband is not acceptable. It is not possible for the voltage to exceed 1.0 volts in an LC circuit of this kind, but if the voltage were to exceed the ideal, the deviation would be still be considered to be zero and there would still be no penalty for a filter design problem.

The procedure for each of the 35 points in the interval from 2,000 Hz to 100,000 Hz is as follows: If the voltage is between 0 millivolts and 1 millivolt, the absolute value of the deviation from 0 millivolts is weighted by a factor of 1.0. If if the voltage is more than 1 millvolt, the absolute value of the deviation from 0 millivolts is weighted by a factor of 10.0. This arrangement reflects the fact that the ideal voltage in the stopband is

0.0 volt, the fact that a 1 millivolt ripple above 0 millvolts is acceptable, and the fact that a voltage above 1 millivolt in the stopband is not acceptable.

We considered the number of fitness cases (61 and 35) in these two main bands to be sufficiently close that we did not attempt to equalize the weight given to the differing numbers of fitness cases in these two main bands.

The deviation is considered to be zero for each of the 5 points in the interval above 1,000 Hz and below 2,000 Hz (i.e., the "don't care" band).

Hits are defined as the number of fitness cases for which the voltage is acceptable or ideal or which lie in the "don't care" band. Thus, the number of hits ranges from a low of 5 to a high of 101 for this problem.

Some of the bizarre circuits that are randomly created for the initial random population and that are created by the crossover operation and the mutation operation in later generations cannot be simulated by SPICE. Circuits that cannot be simulated by SPICE are assigned a high penalty value of fitness (10^8). These circuits become the worst-of-generation circuits for each generation. The practical effect of this high penalty value of fitness is that these individuals are rarely selected to participate in genetic operations and that they quickly disappear from the population.

The population size, M, is 320,000. Since this problem runs slowly, we set the maximum number of generations, G, to a large number and awaited developments. The percentage of genetic operations on each generation was 89% crossovers, 10% reproductions, and 1% mutations. A maximum size of 200 points was established for each of the two result-producing branches in each overall program. The other parameters for controlling the runs of genetic programming were the default values specified in Koza 1994 (appendix D).

This problem was run on a medium-grained parallel Parystec computer system consisting of 64 Power PC 601 80 MHz processors arranged in a toroidal mesh with a host PC Pentium type computer. The so-called *distributed genetic algorithm* for parallelization was used with a population size of$Q$ = 10,000 at each of the D = 64 demes. On each generation, four boatloads of emigrants, each consisting of B = 2% (the migration rate) of the node's subpopulation (selected on the basis of fitness) were dispatched to the four toroidally adjacent processing nodes. See Andre and Koza 1996.

11. Results for the Problem of Designing a Lowpass LC Filter

We present the results of three different runs of genetic programming on the problem of designing the lowpass LC filter.

11.1. FIRST RUN

A run of genetic programming for this problem starts with the random creation of an initial population of 320,000 program trees (each consisting of two result-producing branches) composed of the functions and terminals identified above and in accordance with the syntactic constraints described above.

For each of the 320,000 program trees in the population, the sequence of circuit-constructing functions in the program tree is applied to the common embryonic circuit for this problem (figure 1) in order to create a circuit. The netlist for the resulting circuit

is then determined. This netlist is wrapped inside an appropriate set of SPICE commands and the circuit is then simulated using our modified version of SPICE.

The initial random population of a run of genetic programming is a blind random search of the search space of the problem. As such, it provides a baseline for comparing the results of subsequent generations.

The best circuit of the 320,000 circuits from generation 0 had a fitness of 58.71 (on the scale of weighted volts described earlier) and scored 51 hits. The first result-producing branch of this program tree has 25 points (i.e., functions and terminals) and is shown below:

```
(C (- 0.963 (- (- -0.875 -0.113) 0.880)) (series (flip end) (series  (flip end) (L  -0.277 end) end) (L (- -0.640 0.749) (L -0.123 end))))
```

The second result-producing branch has 5 points and is shown below:

```
(flip (nop (L -0.657 end))))
```

Figure 5 presents this best-of-generation program tree as a rooted, point-labeled tree with ordered branches. The first result-producing branch is rooted at the `C` function (labeled 2) and the second result-producing branch is rooted at the `FLIP` function (labeled 3).

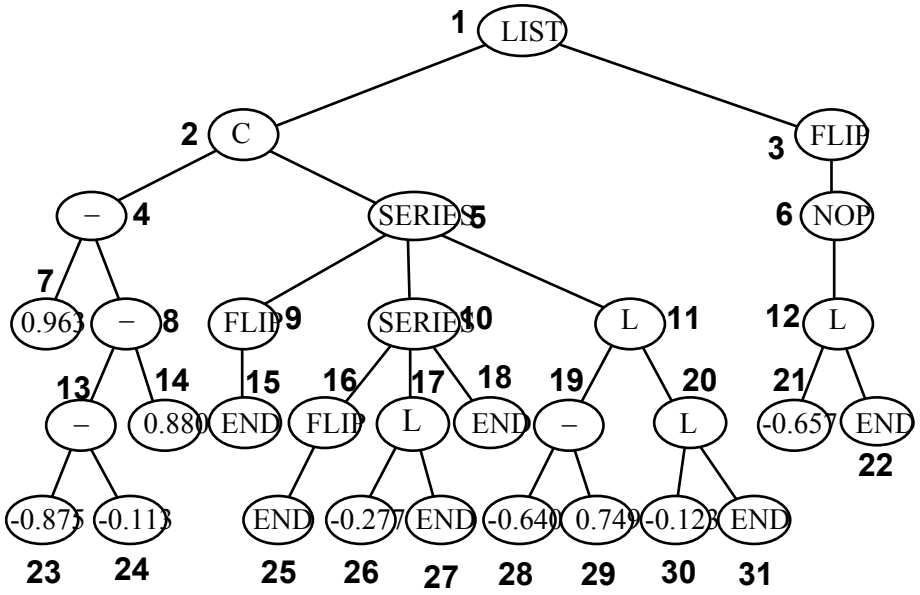

Figure 5 Program tree for best circuit of generation 0.

In executing the program tree, the connective `LIST` function (labeled 1) at the root of the tree is ignored. Most of the remainder of the tree is executed in a breadth-first order; however, arithmetic-performing subtrees (such as the 7-point subtree rooted at the point labeled 4) are executed in their entirety in a depth-first order immediately when its circuit-constructing function is first encountered. Thus, the `C` (capacitor) function (labeled 2) in figure 5 is executed first. Then, the 7-point arithmetic-performing subtree (labeled 4) is immediately executed in its entirety in a depth-first way so as to deliver the

numerical component value needed by the capacitor function `C.` Then, the breadth-first order is resumed and the `FLIP` function (labeled 3) is executed.

Figure 6 shows the best circuit of generation 0 upon completion of the developmental process.

In the frequency domain, the voltages produced by this circuit in the interval between 1 Hz and 100 Hz are very close to the required 1 volt (accounting for most of the 51 hits scored by this individual). However, the voltages produced between 100 Hz and 1,000 Hz deviate considerably below the minimum of 970 millivolts required by the design specification (in fact, by hundreds of millivolts as one approaches 1,000 Hz). Moreover, the voltages produced above 2,000 Hz are, for the most part, considerably above the minimum of 1 millivolt required by the design specification (by hundreds of millivolts in most cases).

Generation 1 (and each subsequent generation of the run) is created from the population at the preceding generation by performing 142,400 crossover operations (producing 284,800 offspring or 89% of 320,000), 32,000 reproduction operations (10% of 320,000), and 3,200 mutation operations (1% of 320,000).

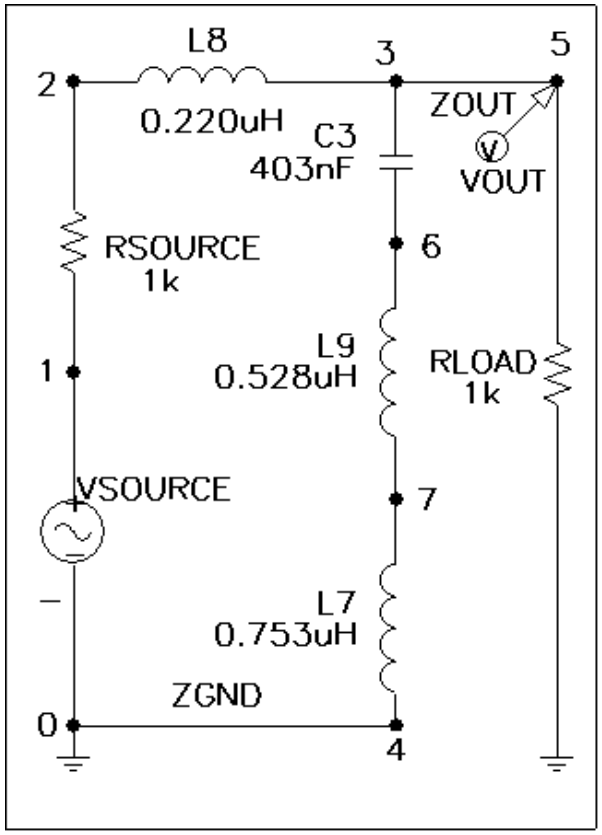

Figure 6 Best circuit of generation 0.

As the run proceeds from generation to generation, the fitness of the best-of-generation individual tends to improve. Figure 7 shows the standardized fitness and number of hits for the best-of-generation program of each generation of this run.

SPICE cannot simulate many of the bizarre circuits created by genetic programming. About two-thirds (65.3%) of the 320,000 programs of generation 0 for this problem produce circuits that cannot be simulated by SPICE. However, the percentage of unsimulatable circuits changes rapidly as new offspring are created by genetic programming using Darwinian selection, crossover, and mutation. The percentage of unsimulatable programs drops to 33% by generation 10, and 0.3% by generation 30. Figure 8 shows, by generation, the percentage of unsimulatable programs in this run.

In the genetic algorithm, the entire population generally improves from generation to generation. The hits histogram is a useful monitoring tool for visualizing the progressive learning of the population as a whole during a run. The horizontal axis of the hits histogram represents the number of hits (0 to 101 here) while the vertical axis represents the percentage of individuals in the population scoring that number of hits.

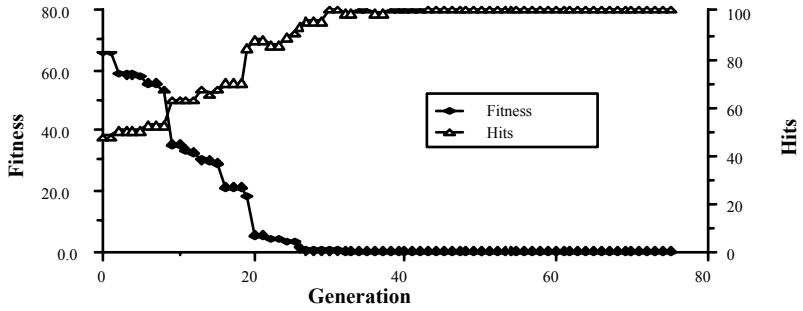

Figure 7 Fitness and hits for one run.

Figure 9 shows the hits histograms for generations 0, 20 and 40 of a typical run of this problem. The horizontal axis represents the number of hits (0 to 101 here) while the vertical axis represents the percentage of individuals in the population scoring that number of hits. Note the left-to-right undulating movement of both the high point and the center of mass of these histograms over the generations.

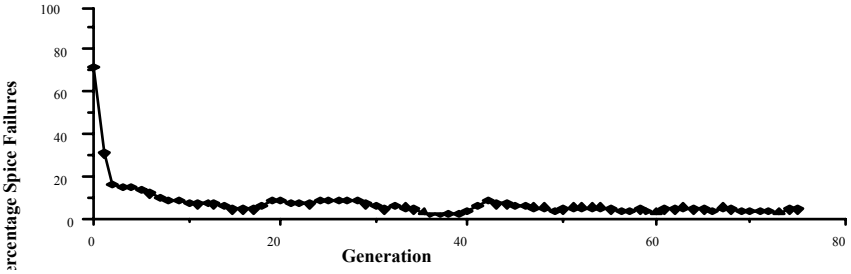

Figure 8 Percentage of unsimulatable programs

The improvement, from generation to generation, in the fitness of the population as a whole can also be seen by examining the average fitness of the population by generation. Figure 10 shows, by generation, the average fitness of the portion of the population that can be analyzed by SPICE (that is, after excluding individuals receiving the penalty value of fitness). As can be seen, the average fitness of the population as a whole is 1,054 for generation 0, 443 for generation 2, 213 for generation 5, 58.2 for generation 10, 38.0 for generation 20, and 16.5 by generation 30.

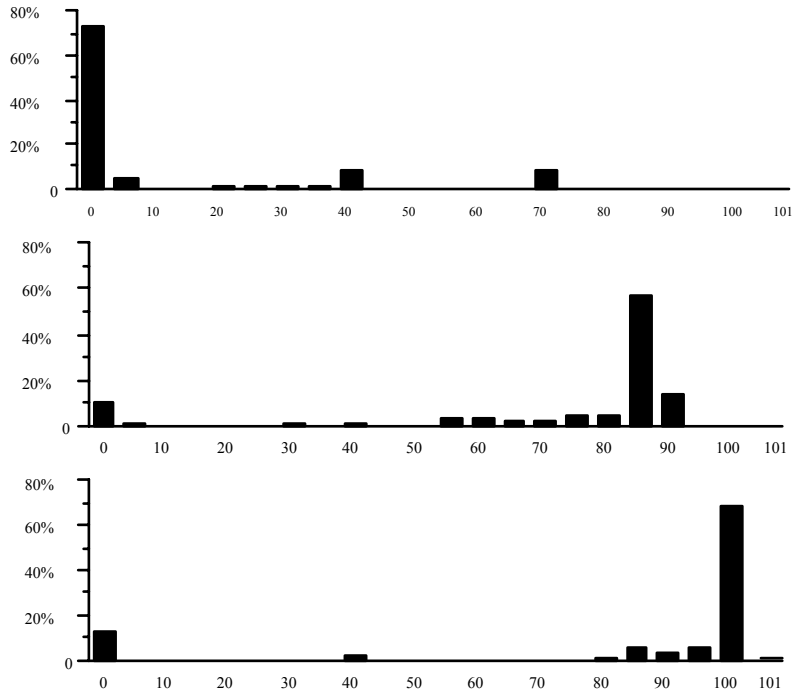

Figure 9 Hits histogram for generations 0, 20 and 40 of a run of this problem.

The best individual program tree of generation 32 has 306 points, has a fitness of 0.00781 and scores 101 hits. That is, by generation 32, all 101 sample points are in compliance with the design requirements for this problem.

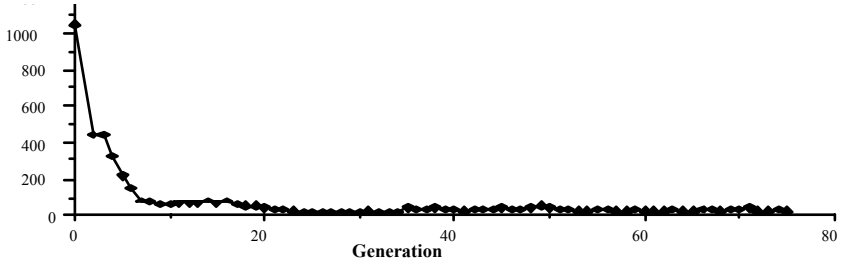

Figure 10 Average fitness of the simulatable circuits in the population.

Figure 11 shows the best-of-run circuit from generation 32. This circuit is a seven-rung ladder consisting of repeated values of various inductors and capacitors.

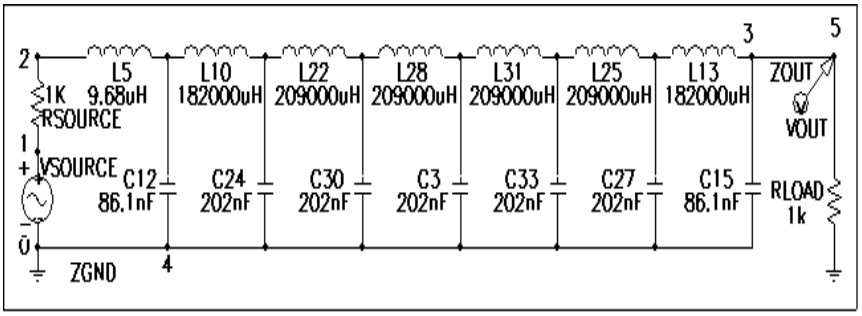

Figure 11 Best-of-run "seven-rung ladder" circuit from generation 32.

Figure 12 shows the behavior in the frequency domain of the best-of-run circuit from generation 32. As can be seen, the circuit delivers a voltage of virtually 1 volt in the entire passband from 1 Hz to 1,000 Hz and delivers a voltage of virtually 0 volts in the entire stopband starting at 2,000 Hz.

The best individual from generation 76 has a fitness (0.000995) that is about an order of magnitude better than that of the fully compliant individual of generation 32.

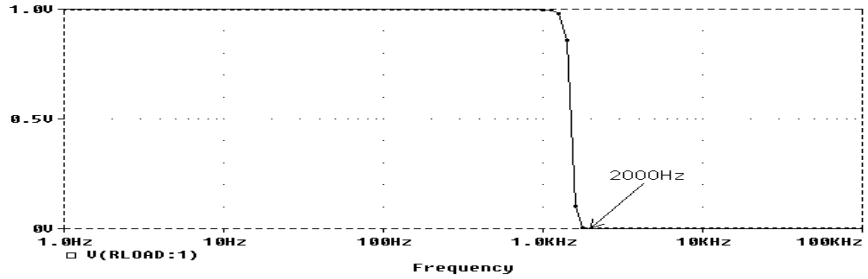

Figure 12 Frequency domain behavior of "seven-rung ladder" from generation 32.

11.2. A "BRIDGED T" CIRCUIT FROM ANOTHER RUN

Different runs of genetic programming produce different results. Moreover, when we continue the run of genetic programming after the emergence of the first 100%-compliant individual, additional 100%-compliant individuals often emerge. Figure 13 shows a fully compliant best-of-run circuit from generation 64 of another run. In this circuit (which has a fitness of 0.04224), inductor L14 forms a "bridged T" subcircuit in conjunction with capacitors C3 and C15 and inductor L11. Of course, the parallel capacitors (the pair C18 and C33 as well as the triplet C24, C21, and C12) could be combined. This "bridged T" circuit is distinctly different in structure from the "ladder" circuit.

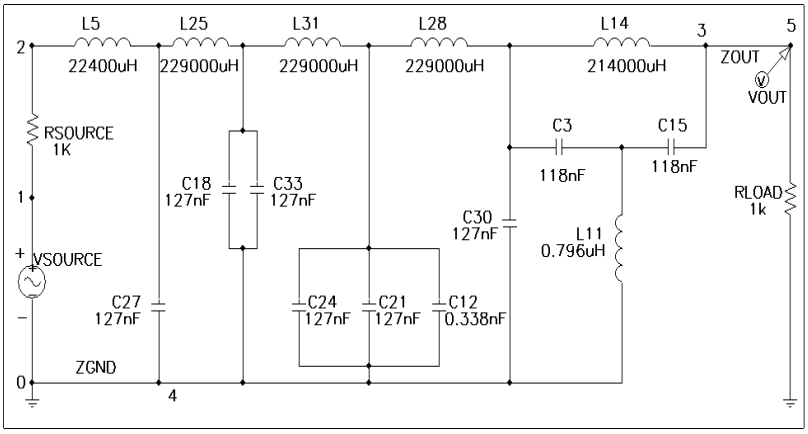

Figure 13 "Bridged T" circuit from generation 64.

12. Subsequent Work and Future Work

We have also used this technique to design an asymmetric bandpass filter and a crossover (woofer and tweeter) filter. The later requires a one-input, two-output embryonic circuit. We are currently working on circuits with active elements.

13. Conclusions

We have described an automated design process for designing analog electrical circuits based on the principles of natural selection, sexual recombination, and developmental biology. The design process starts with the random creation of a large population of program trees composed of circuit-constructing functions. Each program tree specifies the steps by which a fully developed circuit is to be progressively developed from a common embryonic circuit appropriate for the type of problem that the user wishes to solve. The population of program trees is genetically bred over a series of many generations using genetic programming that is driven by the fitness measure. Genetic programming employs genetic operations such as Darwinian reproduction, sexual recombination (crossover), and occasional mutation to create offspring. The paper described how genetic programming technique evolved the design of a low-pass filter.

Acknowledgements

Tom L. Quarles of Meta-Software of Campbell, California provided helpful advice concerning SPICE. Simon Handley made helpful comments on the above.

Bibliography

Aaserud, O. and Nielsen, I. Ring. 1995. Trends in current analog design: A panel debate. *Analog Integrated Circuits and Signal Processing*. 7(1) 5-9.

Andre, David and Koza, John R. 1996. Parallel genetic programming: A scalable implementation using the transputer architecture. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Degrauwe, M. 1987. IDAC: An interactive design tool for analog integrated circuits. *I1 Journal of Solid State Circuits*. 22:1106–1116.

Gruau, Frederic. 1992. *Cellular Encoding of Genetic Neural Networks*. Technical report 92-21. Laboratoire de l'Informatique du Parallélisme. Ecole Normale Supérieure de Lyon. May 1992.

Gruau, Frederic. 1994. Genetic micro programming of neural networks. In Kinnear, Kenneth E. Jr. (editor). *Advances in Genetic Programming*. Cambridge, MA: The MIT Press. Pages 495–518.

Harjani, R., Rutenbar, R. A., and Carley, L. R. 1989. OASYS: A framework for analog circuit synthesis. *I1 Transactions on Computer Aided Design*. 8:1247–1266.

Higuchi, T., Niwa, T., Tanaka, H., Iba, H., de Garis, H. and Furuya, T. 1993. Evolvable hardware – Genetic-based generation of electric circuitry at gate and hardware description language (HDL) levels. Electrotechnical Laboratory technical report 93-4, Tsukuba, Ibaraki, Japan.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press. Second edition. Cambridge, MA: The MIT Press 1992.

Koh, H. Y., Sequin, C. H. and Gray, P. R. 1990. OPASYN: A compiler for MOS operational amplifiers. *I1 Transactions on Computer Aided Design*. 9:113–125.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, John R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Kruiskamp, Wim and Leenaerts, Domine. 1995. DARWIN: CMOS opamp synthesis by means of a genetic algorithm. *Proceedings of the 32nd Design Automation Conference*. New York, NY: Association for Computing Machinery. Pages 433–438.

Maulik, P. C. Carley, L. R., and Rutenbar, R. A. 1992. A mixed-integer nonlinear programming approach to analog circuit synthesis. *Proceedings of the 29th Design Automation Conference*. Los Alamitos, CA: I1 Press, Pages 698–703.

Ning, Z., Kole, M., Mouthaan, T., and Wallings, H. 1992. Analog circuit design automation for performance. *Proceedings of the 14th CICC*. New York: I1 Press. Pages 8.2.1–8.2.4.

Quarles, Thomas, Newton, A. R., Pederson, D. O., and Sangiovanni-Vincentelli, A. 1994. *SPICE 3 Version 3F5 User's Manual*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, California. March 1994.

Rutenbar, R. A. 1993. Analog design automation: Where are we? Where are we going? *Proceedings of the l5th I1 CICC*. New York: I1 Press. Pages 13.1.1-13.1.8.

Samuel, Arthur L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development.* 3(3) 210–229.

Zverev, A. I. 1967. *Handbook of Filter Synthesis*. John Wiley.

AUTOMATED DESIGN OF BOTH THE TOPOLOGY AND SIZING OF ANALOG ELECTRICAL CIRCUITS USING GENETIC PROGRAMMING

JOHN R. KOZA
Computer Science Department
Stanford University
Stanford, California 94305 USA
E-MAIL: Koza@CS.Stanford.Edu
PHONE: 415-941-0336
FAX: 415-941-9430
WWW: http://www-cs-faculty.stanford.edu/~koza/

FORREST H BENNETT III
Visiting Scholar
Computer Science Department
Stanford University
Stanford, California 94305 USA
EMAIL: fhb3@slip.net
PHONE: 415-424-0948

DAVID ANDRE
Visiting Scholar
Computer Science Department
Stanford University
860 Live Oak Ave, #4
Menlo Park, CA 94025 USA
EMAIL: andre@flamingo.stanford.edu
PHONE: 415-326-5113
WWW: http://www-leland.stanford.edu/~phred/

MARTIN A. KEANE
Econometrics Inc.
5733 West Grover
Chicago, IL 60630 USA
EMAIL: makeane@ix.netcom.com
PHONE: 312-777-1524