Four Problems for which a Computer Program Evolved by Genetic Programming is Competitive with Human Performance

John R. Koza
Computer Science Dept.
258 Gates Building
Stanford University
Stanford, California 94305
koza@cs.stanford.edu
http://www-cs-faculty.stanford.edu/~koza/

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
fhb3@slip.net

David Andre
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
andre@flamingo.stanford.edu

Martin A. Keane
Econometrics Inc.
5733 West Grover
Chicago, Illinois 60630
makeane@ix.netcom.com

Abstract **– It would be desirable if computers could solve problems without the need for a human to write the detailed programmatic steps. That is, it would be desirable to have a domain-independent automatic programming technique in which "What You Want Is What You Get" ("WYWIWYG" – pronounced "wow-eee-wig").**

Genetic programming is such a technique. This paper surveys three recent examples of problems (from the fields of cellular automata and molecular biology) in which genetic programming evolved a computer program that produced results that were slightly better than human performance for the same problem.

This paper then discusses the problem of electronic circuit synthesis in greater detail. It shows how genetic programming can evolve *both* the topology of a desired electrical circuit and the sizing (numerical values) for each component in a crossover (woofer and tweeter) filter. Genetic programming has also evolved the design for a lowpass filter, the design of an amplifier, and the design for an asymmetric bandpass filter that was described as being difficult-to-design in an article in a leading electrical engineering journal.

I. INTRODUCTION

Automatic programming is one of the central goals of computer science. Paraphrasing Arthur Samuel (1959), the problem of automatic programming concerns the question of

How can computers be made to do what needs to be done, without being told exactly how to do it?

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 books *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992) and *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a) describe an extension to Holland's 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. (See also Koza and Rice 1992 and Koza 1994b).

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.

(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)*Reproduction*: Copy an existing program to the new population.

(ii)*Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts of two existing programs.

(iii) *Mutation*. Create one new offspring program for the new population by randomly mutating a randomly chosen part of one existing 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 programming system for the run. This result may be a solution (or approximate solution) to the problem.

Sections II, III, and IV of this paper briefly describe three examples of problems where genetic programming has produced a result that is slightly better than human performance on the same problem. Section V then discusses, in greater detail, how genetic programming can be used to automate the process of electronic circuit synthesis for a crossover (woofer and tweeter) filter. Section VI then briefly shows a genetically evolved lowpass filter, a difficult-to-design asymmetric bandpass filter, and an amplifier.

II. CELLULAR AUTOMATA

It is difficult to program cellular automata. This is especially true when the desired computation requires global communication and integration of local information across great distances in the cellular space. Various human-written algorithms have appeared in the past two decades for the majority classification task for one-dimensional cellular automata. Genetic programming with automatically defined functions has evolved a rule for this task with an accuracy of 82.326% (Andre, Bennett, and Koza 1996). This level of accuracy exceeds that of the original Gacs-Kurdyumov-Levin (GKL) rule, all other known subsequent human-written rules, and all other known rules produced by automated approaches for this problem. The genetically evolved rule is qualitatively different from all previous rules in that it employs a larger and more intricate repertoire of domains and particles to represent and communicate information in the cellular space.

III. TRANSMEMBRANE DOMAINS

The goal in the transmembrane segment identification problem is to classify a given protein segment (i.e., a subsequence of amino acid residues from a protein sequence) as being a transmembrane domain or non-transmembrane area of the protein (without using biochemical knowledge concerning hydrophobicity typically used by human-written algorithms for this task). Four different versions of genetic programming have been applied to this problem (Koza 1994a, Koza and Andre 1996a, 1996b). The performance of all four versions using genetic programming is slightly superior to that of algorithms written by knowledgeable human investigators.

IV. PROTEIN MOTIFS

Automated methods of machine learning may prove to be useful in discovering biologically meaningful information hidden in the rapidly growing databases of DNA sequences and protein sequences. Genetic programming successfully evolved motifs for detecting the D-E-A-D box family of proteins and for detecting the manganese superoxide dismutase family (Koza and Andre 1996c). Both motifs were evolved without prespecifying their length. Both evolved motifs employed automatically defined functions to capture the repeated use of common subexpressions. The two genetically evolved consensus motifs detect the two families either as well as, or slightly better than, the comparable human-written motifs found in the PROSITE database.

V. AUTOMATED CIRCUIT SYNTHESIS

The problem of circuit synthesis involves designing an electrical circuit that satisfies user-specified design goals. 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 are not as amenable to automation (Rutenbar 1993).

A complete specification of an electrical circuit includes both its topology and the sizing of all its components. The *topology* of a circuit consists of the number of components in the circuit, the type of each component, and a list (i.e., the netlist) of the connections between the leads (interface points) of the components. The *sizing* of a circuit consists of the component value(s) associated with each component.

V.1. Automated Analog Design Tools

Hemmi, Mizoguchi, and Shimohara (1994) and Higuchi et al. (1993) have applied genetic methods to the design of digital circuits using a hardware description language (HDL).

The design of analog circuits and mixed analog-digital circuits has not proved to be as amenable to automation. In DARWIN (Kruiskamp and Leenaerts 1995), CMOS opamp circuits are designed using the genetic algorithm. In DARWIN, the topology of each opamp is picked randomly from a preestablished hand-designed set of 24 topologies in order to ensure that each circuit behaves as an opamp.

V.2. The Mapping between Electrical 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 by genetic programming and the labeled cyclic graphs encountered in the world of electrical circuits.

Electrical circuits are cyclic graphs in which *every* line belongs to a cycle. The primary label on each line identifies the type of electrical component. The secondary label(s), if any, on each line specify the value(s) of the component.

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 cyclic graphs employed in the world of circuits. Developmental biology provides the motivation for this mapping. In *Cellular Encoding of Genetic Neural Networks*, Frederic Gruau (1992) described an innovative 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 of the neurons in the network. In this technique, genetic programming is applied to populations of network-constructing program trees in order to evolve a neural network capable of solving a problem.

The growth process used herein for circuit synthesis begins with a very simple embryonic electrical circuit and builds a more complex circuit by progressively executing the functions in a circuit-constructing program tree. The result is the topology of the circuit, the choice of the type of component that is situated at each location within the topology, and the sizing of all components.

Each program tree can contain (1) connection-modifying functions that modify the topology of the circuit (starting with the embryonic circuit), (2) component-creating functions that insert particular components into locations within the topology of the circuit in lieu of wires (and other components) and whose arithmetic-performing subtrees specify the numerical value (sizing) for each such component, and possibly (3) automatically defined functions.

Program trees conform to a constrained syntactic structure. Each component-creating function in a program tree has zero, one, or more arithmetic-performing subtrees and one or more construction-continuing subtrees. Each connection-modifying function has one or more construction-continuing subtrees. The arithmetic-performing subtree(s) of each component-creating function consists of a composition of arithmetic functions and numerical constant terminals that together yield the numerical value for the component. The construction-continuing subtree specifies how the construction of the circuit is to be continued.

Both the random program trees in the initial population (generation 0) and all 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 using structure-preserving crossover with point typing (Koza 1994a).

The bottom of figure 1 shows the embryonic circuit for a one-input, two-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 source resistor RSOURCE between nodes 1 and 2. There is a modifiable wire (i.e., a wire with a writing head) Z0 between nodes 2 and 3, a second modifiable wire Z1 between nodes 2 and 6, and third modifiable wire Z2 between nodes 3 and 6. There is an isolating wire ZOUT1 between nodes 3 and 4, a voltage probe labeled VOUT1 at node 4, and a fixed load resistor RLOAD1 between nodes 4 and ground. Also, there is an isolating wire ZOUT2 between nodes 6 and 5, a voltage probe labeled VOUT2 at node 5, and a load resistor RLOAD2 between nodes 5 and ground. The resistors are 0.00794 Kilo Ohms. All of the above elements of this embryonic circuit (except Z0, Z1, and Z2) are fixed forever; they are not subject to modification during the process of developing the circuit. Note that little domain knowledge went into this embryonic circuit. Specifically, (1) the embryonic circuit is a circuit, (2) this embryonic circuit has one input and two outputs, and (3) there are modifiable connections Z0, Z1, and Z2 providing full point-to-point connectivity between the one input (node 2) and the two outputs VOUT1 and VOUT2 (nodes 4 and 5).

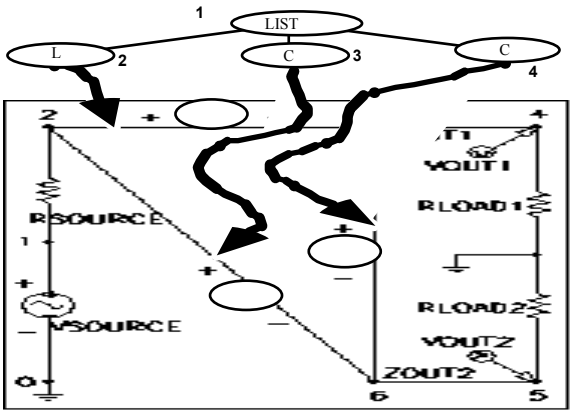

Figure 1 One-input, two-output embryonic electrical circuit.

A circuit is developed by modifying the component to which a writing head is pointing in accordance with the associated function in the circuit-constructing program tree. The figure shows L, C, and C functions just below the LIST and three writing heads pointing to Z0, Z1, and Z2. The L, C, and C functions will cause Z0, Z1, and Z2 to be changed into an inductor and two capacitors, respectively.

V.3. Component-Creating Functions

Each individual circuit-constructing program tree in the population generally contains component-creating functions and connection-modifying functions.

Each component-creating function inserts a component into the developing circuit and assigns component value(s) to the inserted component. Each component-creating function in a program tree points to an associated highlighted component (i.e., a component with a writing head) in the developing circuit and modifies the highlighted component in some way. Each component-creating function spawns one or more writing heads (through its construction-continuing subtrees). The construction-continuing subtree of each component-creating function points to a successor function or terminal in the circuit-constructing program tree.

The arithmetic-performing subtree of a component-creating function consists of a composition of arithmetic functions (addition and subtraction) and random constants (in the range –1.000 to +1.000). The arithmetic-performing subtree specifies the numerical value of the component by returning a floating-point value that is, in turn, interpreted as the value for the component in a range of 10 orders of magnitude (using a unit of measure that is appropriate for the particular type of component involved). The floating-point value is interpreted as the value of the component as

described more fully in Koza, Andre, Bennett, and Keane 1996.

The two-argument capacitor-creating C function causes the highlighted component to be changed into a capacitor. The value of the capacitor in nano Farads is specified by its arithmetic-performing subtree.

The two-argument inductor-creating L function causes the highlighted component to be changed into an inductor. The value of the inductor in micro-Henrys is specified by its arithmetic-performing subtree.

The functions in the group of three-argument transistor-creating QT functions cause a transistor to be inserted in place of one of the nodes to which the highlighted component is connected (while deleting the highlighted component). Each QT function creates five new nodes and three new modifiable wires. After execution of a QTn, there are three writing heads that point to three new modifiable wires. Figure 2 shows a resistor R1 (with a writing head) connecting nodes 1 and 2. Figure 3 shows the result of applying the QT0 function to R1, thereby creating transistor Q6.

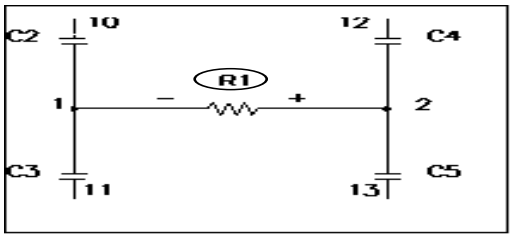

Figure 2 Circuit with resistor R1.

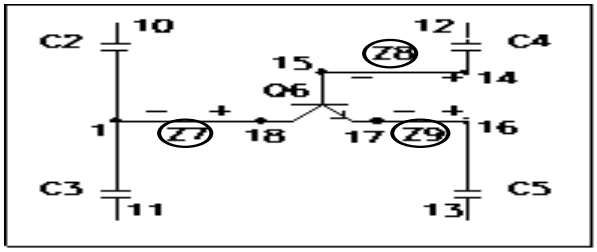

Figure 3 Result of applying QT0 function.

V.4. Connection-Modifying Functions

The topology of the circuit is determined by the connection-modifying functions. Each connection-modifying function in a program tree points to an associated highlighted component and modifies the topology of the developing circuit in some way. Each connection-modifying function spawns zero, one, or more writing heads.

The one-argument polarity-reversing FLIP function attaches the positive end of the highlighted component to the node to which its negative end is currently attached and vice versa. After execution of the FLIP function, one writing head points to the now-flipped original component.

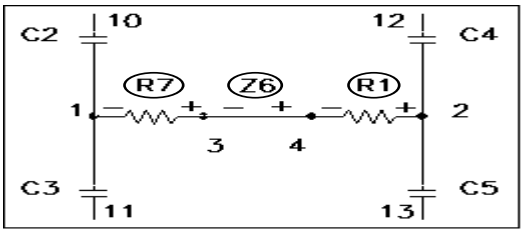

Figure 4 Result of applying SERIES.

The three-argument 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 modifiable 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 4 shows 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 disconnects the negative end of the original component (R1) from node 1 and connects this negative end to the first new node, 4 (while leaving its positive end connected to the node 2). Third, SERIES creates a new wire (called Z6 in the figure) between new nodes 3 and 4. The negative end of the new wire is connected to the first new node 3 and the positive end is connected to the second new node 4. Fourth, SERIES inserts a duplicate (called R7 in the figure) of the original component (including all its component values) between new node 3 and original node 1. The positive end of the duplicate is connected to the original node 1 and its negative end is connected to new node 3.

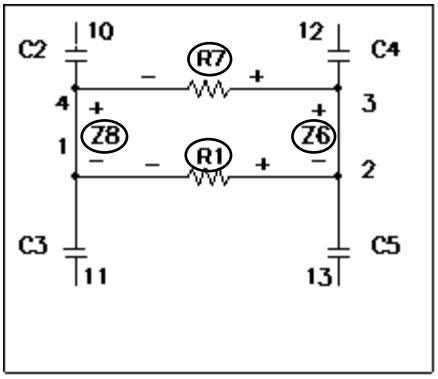

Figure 5 Result of applying PSS.

The four-argument parallel division function PSS operates 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 PSS, there are four writing heads. They point to the original component, the two new modifiable wires, and the copy of the original component. First, the parallel division function PSS creates two new nodes, 3 and 4. Second, PSS 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 3. Third, PSS creates a first new wire Z6 between the positive (+) end of R1 (which is at original node 2) and first new node, 3. Fourth, PSS creates a second new wire Z8 between the negative (-) end of R1 (which is at original node 1) to second new node, 4. Figure 5 shows the results of applying the PSS function to resistor R1 from figure 2. The negative end of the new component is connected to the smaller numbered component of the two components that were originally connected to the negative end of the highlighted component. 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.

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

The VIA 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) of an imaginary silicon wafer. If one or more parts of the circuit connect to a particular layer, all such parts become electrically connected as if wires were running between them.

The two-argument GND function is a special "via" function that establishes a connection directly to ground.

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. After execution of NOP, one writing head points to the original highlighted component.

The zero-argument END function causes the highlighted component to lose its writing head.

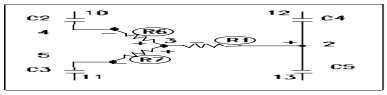

Figure 6 Result of applying the Y1 function.

We describe two other functions (not used herein) to illustrate that numerous other connection-modifying functions can be employed in this process. The functions in the group of three-argument Y division functions operate on one highlighted component (and one adjacent node) and create a Y-shaped composition consisting of the highlighted component, two copies of the highlighted component, and two new nodes. The Y functions insert the two copies at the "active" node of the highlighted component. For the Y1 function, the active node is the node to which the negative end of the highlighted component is connected. Figure 6 shows the result of applying Y1 to resistor R1 of figure 2.

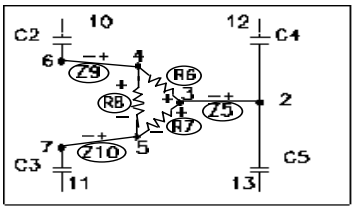

Figure 7 Result of applying DELTA1 function.

The functions in the group of six-argument DELTA functions operate on one highlighted component by eliminating it (and one adjacent node) and creating a triangular Δ–shaped composition consisting of three copies of the original highlighted component (and all of its component values), three new modifiable wires, and five new nodes. Figure 7 illustrates the result of applying the DELTA1 division function to resistor R1 of figure 2 when the active node (node 1) is of degree 3.

V.5. Preparatory Steps for an Example

A *crossover* (woofer and tweeter) filter is a one-input, two-output filter that passes all frequencies below a certain specified frequency to its first output port and that passes all higher frequencies to a second output port. The goal here is to design a crossover filter at a frequency of 2,512 Hertz.

Before applying genetic programming to a circuit synthesis 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, (6) determining the architecture of the overall program, and (7) identifying the embryonic circuit that is suitable for the problem. We first discuss items (7) and (6).

V.5.1 The Embryonic Circuit

The embryonic circuit (figure 1) is suitable for this problem.

V.5.2 Program Architecture

No automatically defined functions are to be used in this problem. Thus, the architecture of the overall program tree consists of three result-producing branches joined by the connective LIST function. Thus, the embryonic circuit initially has three writing heads – one associated with each result-producing branch.

V.5.3 Function and Terminal Sets

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

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

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

$\mathcal{T}_{aps}$ = {←},

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

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

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

The terminal set, $\mathcal{T}_{ccs}$, for the construction-continuing subtree consists of

Tccs = {END}.

V.5.4 Fitness Measure

The evaluation of fitness for each individual circuit-constructing program tree in the population begins with its execution. This execution applies the functions in the program tree to the very simple embryonic circuit thereby developing the embryonic circuit into a fully developed circuit. A netlist describing the circuit is then created. The netlist identifies each component of the circuit, the nodes to which that component is connected, and the value of that component. Each circuit is then simulated to determine its behavior. The 217,000-line SPICE simulator was modified to run as a submodule within the genetic programming system. SPICE (an acronym for Simulation Program with Integrated Circuit Emphasis) is a massive program written over several decades at the University of California at Berkeley for the simulation of analog, digital, and mixed analog/digital electrical circuits. 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 (Quarles et al. 1994).

The fitness measure 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, surface area occupied by its components, or sensitivity to temperature or other variables. Since we are designing a filter, the focus is on the behavior of the circuit in the frequency domain.

The starting point for the design of a filter is the specification by the user of the frequency ranges for its passband and stopband, its maximum *passband ripple* (i.e., the small variation that is tolerated within the passband) and its minimum *stopband attenuation* (i.e., the large degree of blockage of the signal that is demanded in the stopband).

The SPICE simulator is requested to perform an AC small signal analysis and to report the circuit's behavior at two probe points, VOUT1 and VOUT2, for each of 101 frequency values chosen from the range between 10 Hz to 100,000 Hz. Each of these four decades of frequency are divided into 25 parts (using a logarithmic scale) giving 101 fitness cases for each probe point.

Fitness is measured in terms of the sum, over these 101 frequency values, of the absolute weighted deviation between the actual value of voltage in the frequency domain that is produced by the circuit at the first probe point VOUT1 and the target value for voltage for that first probe point *plus* the sum, over these 101 frequency values, of the absolute weighted deviation between the actual value of voltage that is produced by the circuit at the second probe point VOUT2 and the target value for voltage for that second probe point. The smaller the value of fitness, the better. A fitness of zero represents an ideal filter.

Specifically, the standardized fitness, $F(t)$, is

$$F(t) = \sum_{i=0}^{100} \left[W_1(d_1(f_i),f_i)d_1(f_i) + W_2(d_2(f_i),f_i)d_2(f_i) \right]$$

where $f(i)$ is the frequency (in Hertz) of fitness case i; $d1(x)$ is the difference between the target and observed values at frequency x for probe point VOUT1; $d2(x)$ is the difference between the target and observed values at frequency x for probe point VOUT2; $W_1(y,x)$ is the weighting for difference y at frequency x for probe point VOUT1; and $W_2(y,x)$ is the weighting for difference y at frequency x for probe point VOUT2.

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

Consider the woofer (lowpass) portion and VOUT1 first. The procedure for each of the 58 points in the desired passband from 10 Hz to 1,905 Hz is as follows: 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 satisfies the design goals, and the fact that a voltage below 970 millivolts in the passband is not acceptable. For the 38 fitness cases representing frequencies of 3,311 and higher in the intended stopband, the procedure is as follows: If the voltage is between 0 millivolts and 1 millivolts, the absolute value of the deviation from 0 millivolts is weighted by a factor of 1.0. If the voltage is more than 1 millivolts, 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 millivolts is acceptable, and the fact that a voltage above 1 millivolt in the stopband is not acceptable.

For the two fitness cases at 2,089 Hz and 2,291 Hz, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 1.0. For the fitness case at 2,512 Hz, the absolute value of the deviation from 500 millivolts is weighted by a factor of 1.0. For the two fitness cases at 2,754 Hz and 3,020 Hz, the absolute value of the deviation from 0 millivolts is weighted by a factor of 1.0.

The fitness measure for the tweeter (highpass) portion involving VOUT2 is a mirror image of the arrangement for the woofer portion.

Hits are defined as the number (10 to 202) of fitness cases for which the voltage is acceptable.

Some circuits that are randomly created for the initial random population and that are created by the crossover and mutation operations in later generations are so bizarre that they cannot be simulated by SPICE. Circuits that cannot be simulated by SPICE are assigned a high penalty value of fitness (10^8).

V.5.5 Control Parameters

The population size, *M*, is 640,000. The crossover percentage was 89% (producing 569,600 offspring); the reproduction percentage was 10%; and the mutation percentage was 1%. A maximum size of 200 points was established for each of the three result-producing branches in each overall program. The other minor parameters were the default values in Koza 1994a (appendix D).

V.5.6 Parallel Computer System

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* 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.

V.6. Results

The worst individual program trees from generation 0 create circuits that are so pathological that SPICE is incapable of simulating them. The best circuit (figure 8) from generation 0 has a fitness of 159.0 and scores 85 hits (out of 202). Its frequency domain behavior is shown in figure 11.

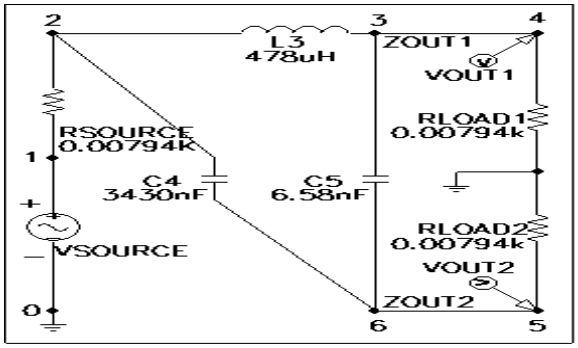

Figure 8 Best circuit of generation 0.

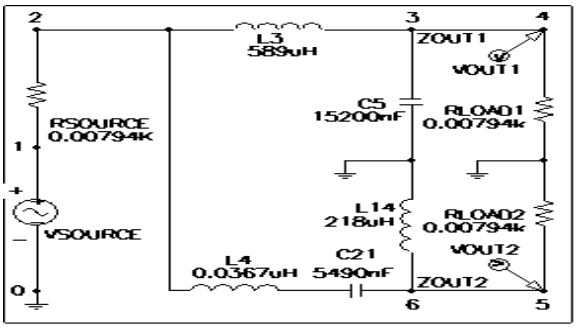

Figure 9 Best circuit of generation 20.

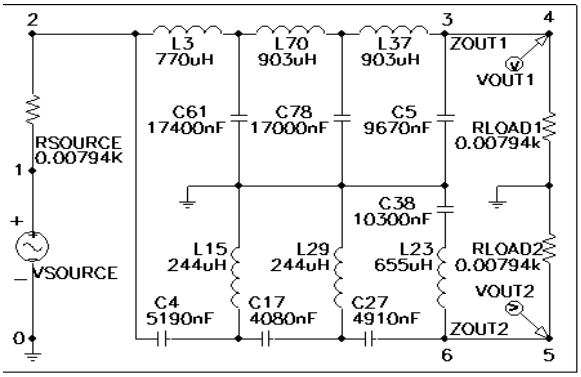

Figure 10 Best-of-run circuit from generation 137.

SPICE cannot simulate many of the bizarre circuits created by genetic programming. About 98.5% of the programs of generation 0 for this problem produce circuits that cannot be simulated by SPICE. However, the percentage of unsimulatable programs drops to 84.9% by generation 1, 75.0% by generation 2, and an average of 9.6% thereafter.

This observation supports the general principle that the individuals in the population in intermediate generations of a run of genetic programming (and random subtrees picked from them) differ markedly from the individuals (and their randomly picked subtrees) in the randomly created population of generation 0 of the same run. That is, crossover fragments from intermediate generations of a run of genetic programming are very different from the randomly grown subtrees provided by the mutation operation. It is experimental evidence, for this non-trivial problem, that the population serves a vital role in the genetic algorithm – namely that of providing a reservoir of useful fragments to rapidly advance the search.

In embarking on this project of trying to evolve electronic circuits using genetic programming, one of our major threshold concerns was whether any significant percentage of the randomly created circuits of generation 0 in this highly epistatic search space would be simulatable at all by SPICE. A second concern was whether the crossover operation would create any significant percentage of simulatable circuits. Neither of these issues materialized on this problem. Darwinian selection apparently is very effective in quickly steering the population on successive generations into the portion of the search space where parents can successful sire simulatable offspring by means of crossover.

The best-of-generation individual from generation 20 (figure 9) has a fitness of 38.8 and scores 125 hits (out of 202). Its frequency domain behavior is shown in figure 12.

The best-of-run circuit (figure 10) from generation 137 has a fitness of 0.7807 and 192 hits (out of 202). Its frequency domain behavior is shown in figure 13.

V.7. Comparison with Butterworth Filters

The Butterworth filters are a graded series of benchmark "ladder" filters parameterized by *n*, where *n* is the number of inductors and capacitors in the circuit.

Figure 14 shows the frequency domain response of this combination of two Butterworth filters of order 3. When we apply our fitness and hits measures to a combination of lowpass and highpass Butterworth 3 filters, the combined circuit scores 162 hits (out of 202). Figure 15 shows the frequency domain response of two Butterworth 5 filters (which corresponds to a score of 184 hits) and figure 16 shows two Butterworth 7 filters (with 190 hits).

The best-of-run circuit from generation 137 described above scores 192 hits and thus can be said to deliver a response that is slightly better than the combination of lowpass and highpass Butterworth filters of order 7.

The lowpass part of the best-of-run circuit has the same topology as Butterworth (but not the Butterworth component values). The highpass part has an extra capacitor and a sharper boundary around the crossover frequency of 2,512 Hz.

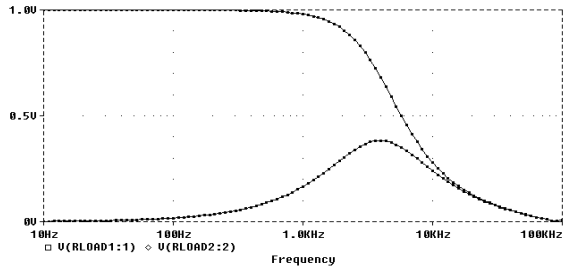

Figure 11 Frequency domain behavior of the best circuit of generation 0.

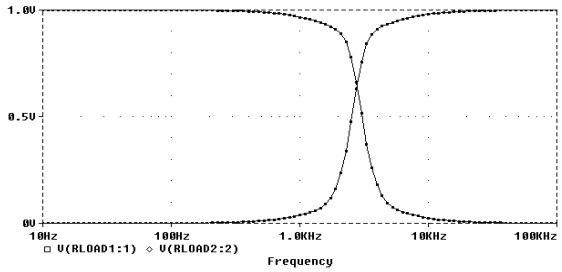

Figure 12 Frequency domain behavior of the best of generation 20.

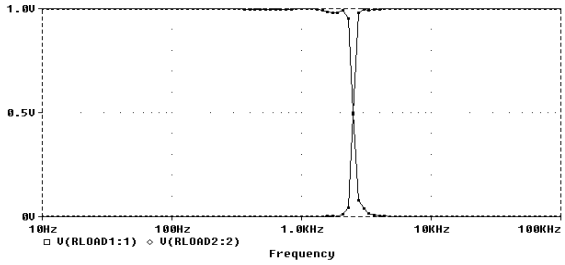

Figure 13 Frequency domain behavior of the best-of-run circuit from generation 137.

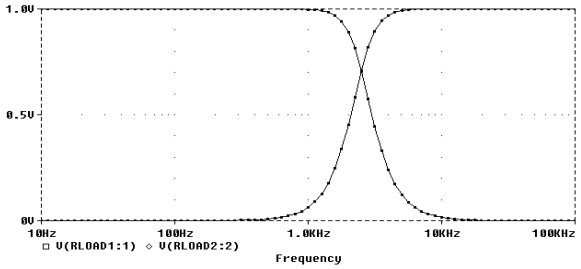

Figure 14 Frequency domain behavior of two Butterworth 3 filters.

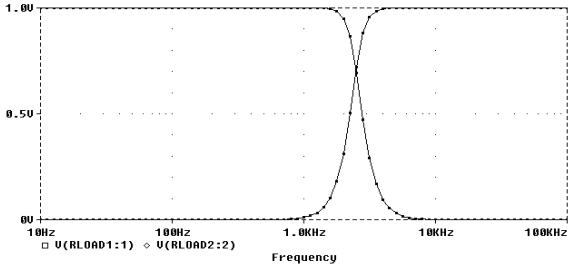

Figure 15 Frequency domain behavior of two Butterworth 5 filters.

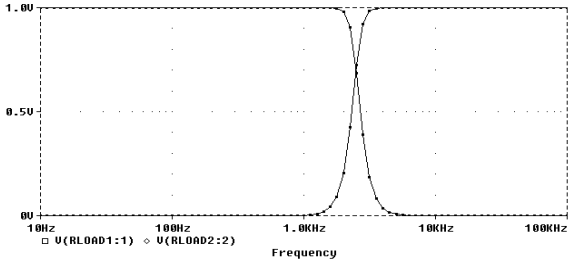

Figure 16 Frequency domain behavior of two Butterworth 7 filters.

VI. OTHER GENETICALLY EVOLVED CIRCUITS

The above techniques have recently been successfully applied to a variety of other problems of circuit synthesis.

VI.1. Lowpass "Brick Wall" Filter

Consider the problem of designing a lowpass filter with passband below 1,000 Hz and a stopband above 2,000 Hz (as described more fully in Koza, Bennett, Andre, and Keane 1996). The voltages in the passband are to be between 970

millivolts and 1 volt (i.e., the passband ripple is 30 millivolts) and the voltages in the stopband between 0 volts and 1 millivolt. Figure 17 shows the 100% compliant circuit that was evolved in one run. This genetically evolved lowpass filter has a recognizable "ladder" topology of a Butterworth or Chebychev filter and consists of a series composition inductors with capacitors as shunts.

In another run, a 100% compliant recognizable "bridged T" arrangement was evolved (involving capacitors C3 and C15 and inductor L11 in conjunction with L14), as shown in figure 18.

Figure 19 shows a 100% compliant circuit from generation 212 of another run with a novel topology that no electrical engineer would be likely to create.

VI.2. An Asymmetric Bandpass Filter

In the *Analog Integrated Circuits and Signal Processing* journal, Nielsen (1995) presented specifications for a difficult-to-design asymmetric bandpass filter. Using a standard bandpass filter on his problem would require a tenth-order elliptic function.

Nielsen's bandpass filter (1995) is targeted for a modem application where one band of frequencies (31.2 to 45.6 kilohertz) must be isolated from another (69.6 to 84.0 KHz). Nielsen specifies that it would be *ideal* if the relative voltage within the passband were in the narrow region between –0.6 dB and 0.6 dB (i.e., the *passband ripple* around 0 dB is less than 0.6 decibels) and all the relative voltages outside the passband were below –120 dB (i.e., the *stopband attenuation* were at least 120 dB). These ideal characteristics are depicted by the dark region in figure 21 (which exaggerates the height of the ripple band). Nielsen also defined a set of *acceptable* characteristics (depicted by the light shading in figure 21). Because of the importance of isolating the band of frequencies between 69.6 and 84.0 KHz, the attenuation there should be at least 73 dB (i.e., the relative voltage is below –73 dB).

Less stringency is demanded elsewhere. The attenuation for frequencies below 20 KHz should be at least 38 dB (i.e., the relative voltage is below –38 dB). The relative voltages in the frequency band between 20 KHz and 31.2 KHz and in the band between 45.6 KHz and 69.6 KHz should be below 0 dB. The relative voltages in the band above 84.0 KHz should be below –55 dB.

In one run (as described more fully in Koza, Bennett, Andre, and Keane 1996), several fully compliant circuits were evolved between generations 132 and 199. Figure 20 shows the best-of-run circuit from generation 199 and figure 21 shows its behavior in the frequency domain.

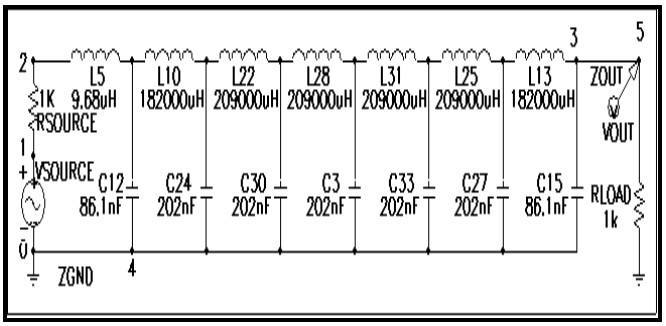

Figure 17 Seven-rung ladder circuit from generation 32.

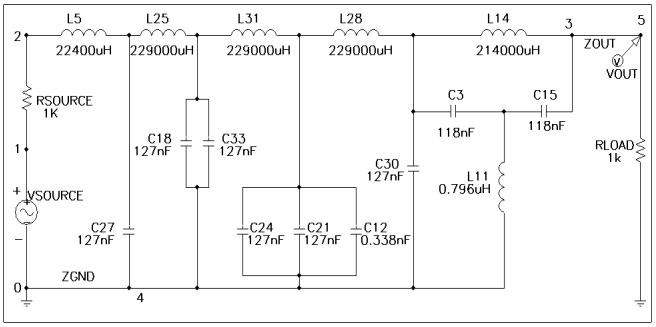

Figure 18 "Bridged T" from generation 64.

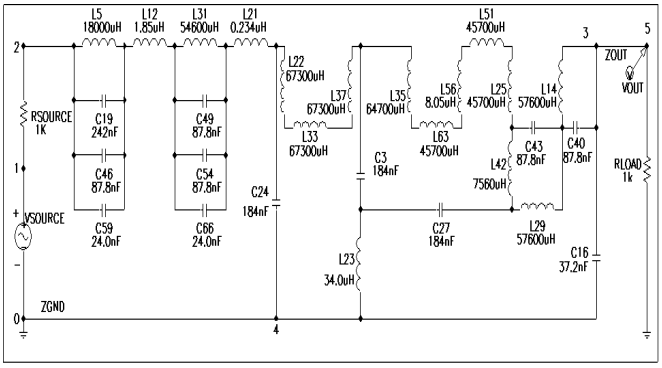

Figure 19 Best circuit from generation 212.

VI.3. An Amplifier using Transistors

Now consider the problem of designing an amplifier with an amplification factor of 3.5 over the frequency range of 20 Hz to 20,000 Hz. Amplifiers are active circuits and require active components (e.g., transistors) in the function set. Figure 22 shows a genetically evolved 5 dB amplifier. The boxes highlight a recognizable voltage gain stage and a recognizable Darlington emitter follower section (inverted).

VII. CONCLUSION

We have surveyed four fields in which genetic programming has evolved computer programs that are competitive in performance with human-written programs.

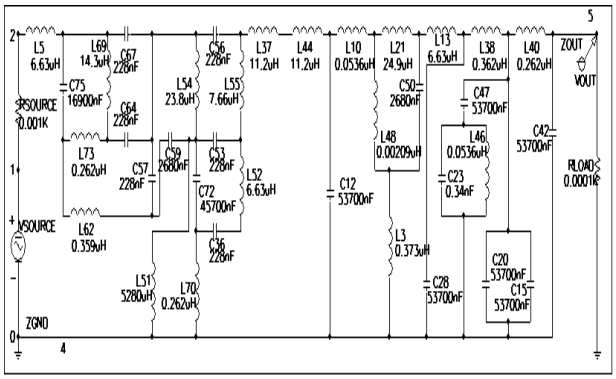

Figure 20 Best-of-run circuit of generation 199.

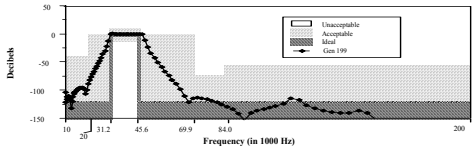

Figure 21 Frequency domain behavior of best-of-run circuit of generation 199.

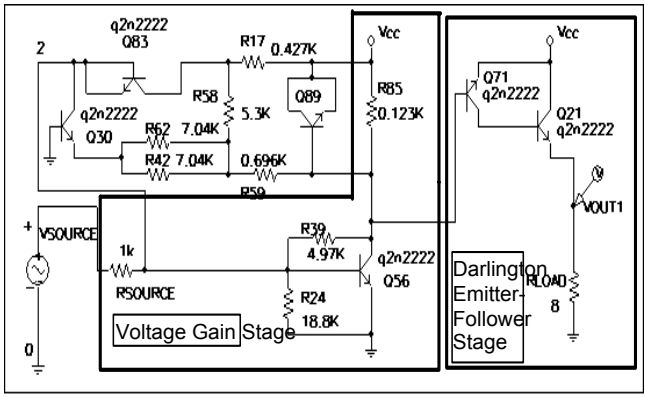

Figure 22 Genetically evolved 5 dB amplifier from generation 45.

BIBLIOGRAPHY

Andre, David, Bennett III, Forrest H, and Koza, John R. 1996. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. In Press.

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

Degrauwe, M. 1987. IDAC: An interactive design tool for analog integrated circuits. *IEEE 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.

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

Hemmi, Hitoshi, Mizoguchi, Jun'ichi, and Shimohara, Katsunori. 1994. Development and evolution of hardware behaviors. In Brooks, R. and Maes, P. (editors). *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: MIT Press. Pages 371–376.

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 System*. Ann Arbor, MI: University of Michigan Press.

Koh, H. Y., Sequin, C. H. and Gray, P. R. 1990. OPASYN: A compiler for MOS operational amplifiers. *IEEE 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: MIT Press.

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

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R. and Andre, David. 1996a. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming II*. Cambridge, MA: MIT Press. In Press.

Koza, John R. and Andre, David. 1996b. Evolution of iteration in genetic programming. In *Evolutionary Programming V*: *Proceedings of the Fifth Annual Conference on Evolutionary Programming*. Cambridge, MA: MIT Press. In Press.

Koza, John R. and Andre, David. 1996c. Automatic discovery of protein motifs using genetic programming. In Yao, Xin (editor). 1996. *Evolutionary Computation: Theory and Applications*. Singapore: World Scientific. In Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996. Automated WYWIWYG design of both the topology and component values of analog electrical circuits using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Proceedings of the Fourth*

International Conference on Artificial Intelligence in Design. Kluwer. In Press.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: 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: IEEE Press. Pages 698–703.

Mizoguchi, Junichi, Hemmi, Hitoshi, and Shimohara, Katsunori. 1994. Production genetic algorithms for automated hardware design through an evolutionary process. *Proceedings of the First IEEE Conference on Evolutionary Computation.* IEEE Press. Vol. I. 661-664.

Nielsen, Ivan Riis. 1995. A C-T filter compiler - From specification to layout. *Analog Integrated Circuits and Signal Processing*. 7(1):21–33.

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