Design of a 96 Decibel Operational Amplifier and Other 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

David Andre
Computer Science Dept.
University
of California
Berkeley, California
dandre@cs.berkeley.edu

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

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 (one from the field of cellular automata and two from the fields of 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 a fourth problem in greater detail and demonstrates that a design for a low-distortion 96 decibel op amp (including both topology and component sizing) can be evolved using genetic programming. The information that the user must supply to genetic programming consists of the parts bin (transistors, resistors, and capacitors) and the fitness measure for the major operating characteristics of an op amp.

1. 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* (GA).

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992) describes an extension of Holland's genetic algorithm in which the population consists of computer programs of varying size and shapes (that is, compositions of primitive functions and terminals). Genetic programming (GP) 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.

The book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a, 1994b) describes multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms (called *automatically defined functions*).

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.

Kinnear (1994), Angeline and Kinnear (1996), Koza, Goldberg, Fogel, and Riolo (1996) each contain numerous recent research papers on genetic programming. Gruau (1996) applies genetic programming to evolve neural networks.

Sections 2, 3, and 4 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 5 discusses, in greater detail, how genetic programming can be used to automate the process of electronic circuit synthesis for a 96 decibel operational amplifier with low distortion. The information that the user must supply to genetic programming is minimal and consists of the contents of the parts bin (transistors, resistors, and capacitors) and the fitness measure for the major operating characteristics of an op amp.

2. Discovery of a Cellular Automata Rule that is Better than any Known Rule for the Majority Classification Problem Using Genetic Programming

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.

3. Classifying Protein Segments as 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.

4. Automatic Discovery of 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.

5. Design of a 96 Decibel Operational Amplifier

5.1. The Problem of Automated Circuit Synthesis

The problem of circuit synthesis involves designing an electrical circuit that satisfies user-specified design goals. The design of analog circuits and mixed analog-digital circuits has not proved to be amenable to automation (Rutenbar 1993). Evolvable digital hardware (Hemmi, Mizoguchi, and Shimohara (1994); Mizoguchi, Hemmi, and Shimohara 1994; Higuchi et al. (1993); and Sanchez and Tomassini 1996) offers a potential approach to automated circuit synthesis. Thompson (1996) used a genetic algorithm to evolve a frequency discriminator on a Xilinx 6216 reconfigurable gate array in analog mode. Kruiskamp and Leenaerts (1995) evolved a design for a CMOS op amp circuit using a genetic algorithm; however, the topology of each op amp was one of 24 user-supplied topologies that reflected the conventional human-designed stages of an op amp.

5.2. Evolution of Circuits from an Embryonic Circuit Using Circuit-Constructing Program Trees

In nature, the design of complex structures is accomplished by means of evolution and natural selection. This suggests the possibility of applying the techniques of evolutionary computation to design problems.

Genetic programming can be applied to circuits if a mapping is established between the kind of rooted, point-labeled trees with ordered branches found in the world of genetic programming and the line-labeled cyclic graphs encountered in the world of circuits.

Developmental biology provides the motivation for this mapping. The starting point of the growth process used herein is a very simple embryonic electrical circuit. The embryonic circuit contains modifiable wires along with certain fixed parts appropriate to the design problem at hand. An electrical circuit is progressively developed by applying the functions in a circuit-constructing program tree to the embryonic circuit (and its successor circuits).

The functions in the circuit-constructing program trees are divided into four categories:

(1) connection-modifying functions that modify the topology of circuit, and

(2) component-creating functions that insert components into the topology of the circuit,

(3) arithmetic-performing functions that appear in arithmetic-performing subtrees as argument(s) to the component-creating functions and that specify the numerical value of the component, and

(4) automatically defined functions that appear in function-defining branches and potentially enable certain substructures to be reused.

Each branch of the program tree is created in accordance with a constrained syntactic structure. Branches are composed from construction-continuing subtrees that continue the developmental process and arithmetic-performing subtrees that determine the numerical value of components. Connection-modifying functions have one or more construction-continuing subtrees, but no arithmetic-performing subtrees. Component-creating functions have one construction-continuing subtree and typically have one arithmetic-performing subtree. This constrained syntactic structure is preserved by using structure-preserving crossover with point typing (Koza 1994a).

5.2.1. The Embryonic Circuit

The developmental process for converting a program tree into an electrical circuit begins with an embryonic circuit.

Figure 1 shows a one-input, one-output embryonic circuit that serves as a test harness for evolving op amp circuits. VSOURCE is the incoming signal. VOUT is the output signal. There is a fixed 1,000 Ohm load resistor RLOAD and a fixed 100 Ohm source resistor RSOURCE. Because we are evolving an amplifier, there is also a fixed 100,000,000 Ohms feedback resistor RFEEDBACK, a fixed 100 Ohm balancing source resistor RBALANCE_SOURCE, and a fixed 100,000,000 Ohm balancing feedback resistor RBALANCE_FEEDBACK. This arrangement limits the possible amplification of the evolving circuit to the 1,000,000-to-1 ratio (which corresponds to 120 dB) of the feedback resistor to the source resistor.

There are three modifiable wires Z0, Z1, and Z2 arranged in a triangle so as to provide connectivity between the input, the output, and ground. All of the above elements (except Z0, Z1, and Z2) are fixed and not modified during the developmental process. At the beginning of the developmental process, there is a writing head pointing to (highlighting) each of the three modifiable wires. All development occurs at wires or components to which a writing head points.

The top part of this figure shows a portion of an illustrative circuit-constructing program tree. It contains a resistor-creating R function (labeled 2 and described later), a capacitor-creating C function (labeled 3), and a polarity-reversing FLIP function (labeled 4) and a connective LIST function (labeled 1). The R and C functions cause modifiable wires Z0 and Z1 to become a resistor and capacitor, respectively; the FLIP function reverses the polarity of modifiable wire Z2.

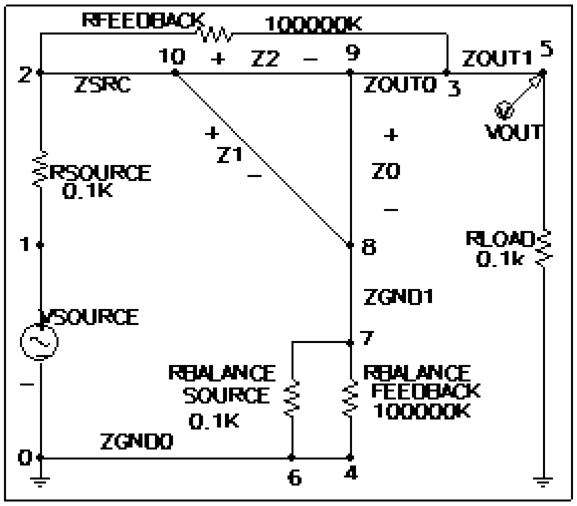

Figure 1 Feedback embryo for an amplifier.

5.2.2. Component-Creating Functions

Each circuit-constructing program tree in the population 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 component. Each component-creating function has a writing head that points to an associated highlighted component in the developing circuit and modifies the highlighted component in some way. 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 a component by returning a floating-point value that is, in turn, interpreted, in a logarithmic way, 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 in Koza, Andre, Bennett, and Keane (1996, 1997)

The two-argument resistor-creating R function causes the highlighted component to be changed into a resistor. The value of the resistor in kilo-Ohms is specified by its arithmetic-performing subtree.

Figure 2 shows a modifiable wire Z0 connecting nodes 1 and 2 of a partial circuit containing four capacitors. Figure 3 shows the result of applying the R function to the modifiable wire Z0 of figure 2. Similarly, 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. Space does not permit a detailed description of each function herein. See Koza, Andre, Bennett, and Keane (1996, 1997), and Koza, Bennett, Andre, and Keane (1996a, 1996b, 1996c, 1996d) for details.

The one-argument Q_D_PNP diode-creating function causes a diode to be inserted in lieu of the highlighted component, where the diode is implemented using a PNP transistor whose collector and base are connected to each other. The Q_D_NPN function inserts a diode using an NPN transistor in a similar manner.

There are also six one-argument transistor-creating functions (called Q_POS_COLL_NPN, Q_GND_EMIT_NPN, Q_NEG_EMIT_NPN, Q_GND_EMIT_PNP, Q_POS_EMIT_PNP, Q_NEG_COLL_PNP) that insert a transistor in lieu of the highlighted component. For example, the Q_POS_COLL_NPN function inserts a NPN transistor whose collector is connected to the positive voltage source.

The three-argument transistor-creating Q_3_NPN function causes an NPN transistor (model Q2N3904) to be inserted in place of the highlighted component and one of the nodes to which the highlighted component is connected. The Q_3_NPN function creates five new nodes and three new modifiable wires. There is no writing head on the new transistor. Similarly, the three-argument transistor-creating Q_3_PNP function causes a PNP transistor (model Q2N3906) to be inserted.

Figure 4 shows the result of applying the Q_3_NPN0 function, thereby creating transistor Q6 in lieu of modifiable wire Z0 of figure 2.

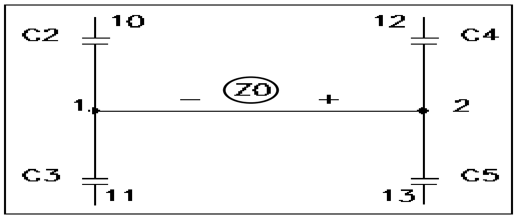

Figure 2 Modifiable wire Z0.

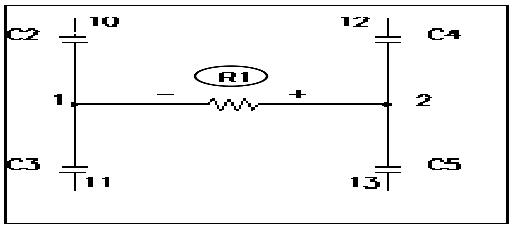

Figure 3 Result of applying the R function.

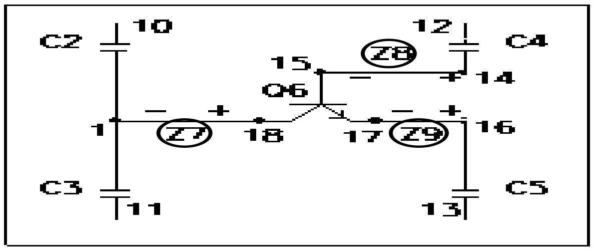

Figure 4 Result of applying Q_3_NPN0 function.

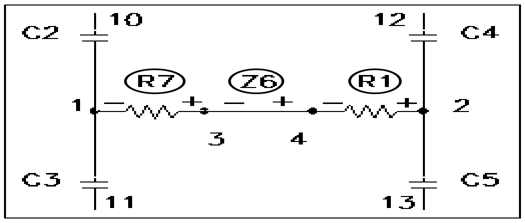

Figure 5 Result after applying the SERIES function.

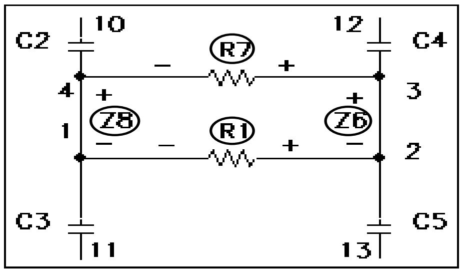

Figure 6 Result of the PSS parallel division function.

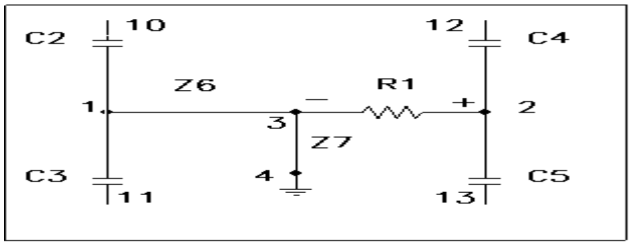

Figure 7 Result of applying the T_GND_0 function.

5.2.3. 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.

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, there is one writing head pointing to the component.

The three-argument SERIES division function creates a series composition consisting of the highlighted component, a copy of it, one new modifiable wire, and two new nodes (each with a writing head).

Figure 5 illustrates the result of applying the SERIES division function to resistor R1 from figure 3.

The four-argument PSS and PSL parallel division functions create a parallel composition consisting of the original highlighted component, a copy of it two new wires, and two new nodes. Figure 6 shows the result of applying PSS to the resistor R1 from figure 3.

There are six three-argument functions (called T_GND_0, T_GND_1, T_POS_0, T_POS_1, T_NEG_0, T_NEG_1) that insert two new nodes and two new modifiable wires and make a connection to ground, positive voltage source, or negative voltage source, respectively. Figure 7 shows the T_GND_0 function connecting resistor R1 to ground.

The three-argument PAIR_CONNECT_0 and PAIR_CONNECT_1 functions enable distant parts of a circuit to be connected together. The first PAIR_CONNECT to occur in the development of a circuit creates two new wires, two new nodes, and one temporary port. The next PAIR_CONNECT to occur (whether PAIR_CONNECT_0 or PAIR_CONNECT_1) creates two new wires and one new node, connects the temporary port (TEMP) to the end of one of these new wires, and then removes the temporary port.

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.

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

The zero-argument SAFE_CUT function causes the highlighted component to be removed from the circuit provided that the degree of the nodes at both ends of the highlighted component is three (i.e., no dangling components or wires are created).

5.3. Preparatory Steps

Our goal in this paper is to evolve the design of a high-gain amplifier. Before applying genetic programming to circuit synthesis, the user must perform seven major preparatory steps, namely

(1) identifying the embryonic circuit that is suitable for the problem,

(2) determining the architecture of the overall circuit-constructing program trees,

(3) identifying the terminals of the to-be-evolved programs,

(4) identifying the primitive functions contained in the to-be-evolved programs,

(5) creating the fitness measure,

(6) choosing certain control parameters (notably population size and the maximum number of generations to be run), and

(7) determining the termination criterion and method of result designation.

The feedback embryo of figure 1 is suitable for this problem.

Since the embryonic circuit has three writing heads – one associated with each of the result-producing branches – there are three result-producing branches (called RPB0, RPB1, and RPB2) in each program tree.

The number of automatically defined functions, if any, will emerge as a consequence of the evolutionary process using the architecture-altering operations.

Each program in the initial population of programs has a uniform architecture with no automatically defined functions (i.e., three result-producing branches).

The terminal sets are identical for all three result-producing branches of the program trees for this problem. The function sets are identical for all three result-producing branches.

The initial function set, $\mathcal{F}_{\text{ccs-initial}}$, for each construction-continuing subtree is

$\mathcal{F}_{\text{ccs-initial}}$ = {R, C, SERIES, PSS, PSL, FLIP, NOP, NEW_T_GND_0, NEW_T_GND_1, NEW_T_POS_0, NEW_T_POS_1, NEW_T_NEG_0, NEW_T_NEG_1, PAIR_CONNECT_0, PAIR_CONNECT_1, Q_D_NPN, Q_D_PNP, Q_3_NPN0, ..., Q_3_NPN11, Q_3_PNP0, ..., Q_3_PNP11, Q_POS_COLL_NPN, Q_GND_EMIT_NPN, Q_NEG_EMIT_NPN, Q_GND_EMIT_PNP, Q_POS_EMIT_PNP, Q_NEG_COLL_PNP}

For the NPN transistors, the Q2N3904 model was used. For PNP transistors, the Q2N3906 model was used.

The initial terminal set, $\mathcal{T}_{\text{ccs-initial}}$, for each construction-continuing subtree is

$\mathcal{T}_{\text{ccs-initial}}$ = {END, SAFE_CUT}.

The set of potential new functions, $\mathcal{F}_{\text{potential}}$, is

$\mathcal{F}_{\text{potential}}$ = {ADF0, ADF1, ADF2, ADF3}.

The set of potential new terminals, $\mathcal{T}_{\text{potential}}$, is

$\mathcal{T}_{\text{potential}}$ = {ARG0}.

The architecture-altering operations change the function set, $\mathcal{F}_{\text{ccs}}$ for each construction-continuing subtree of all three result-producing branches and the function-defining branches, so

$\mathcal{F}_{\text{ccs}} = \mathcal{F}_{\text{ccs-initial}} \approx \mathcal{F}_{\text{potential}}$.

The architecture-altering operations change the terminal set, $\mathcal{T}_{\text{aps-adf}}$, for each arithmetic-performing subtree, so

$\mathcal{T}_{\text{aps-adf}} = \mathcal{T}_{\text{ccs-initial}} \approx \mathcal{T}_{\text{potential}}$.

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

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

where ← represents floating-point random constants from –1.0 to +1.0.

The function set, $\mathcal{F}_{\text{aps}}$, for each arithmetic-performing subtree is,

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

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 embryonic circuit, thereby developing the embryonic circuit into a fully developed circuit. A netlist describing the fully developed 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 (Simulation Program with Integrated Circuit Emphasis) simulation program (Quarles et al. 1994) was modified to run as a submodule within the genetic programming system.

An amplifier can be viewed in terms of its response to a DC input. An ideal amplifier circuit would receive a DC input, invert it, and multiply it by the amplification factor. A circuit is flawed to the extent that it does not achieve the desired amplification; to the extent that the output signal is not centered on 0 volts (i. e., it has a bias); and to the extent that the DC response of the circuit at several different DC input voltages is not linear.

Thus, for this problem, we used a fitness measure based on SPICE's DC sweep. The DC sweep analysis measures the DC response of the circuit at several different DC input voltages. The circuits were analyzed with a 5 point DC sweep ranging from –10 millvolts to +10 MV, with input points at –10 MV, –5 MV, 0 MV, +5 MV, and +10 MV. SPICE then produced the circuit's output for each of these five DC input values.

Fitness is then calculated from four penalties derived from these five DC output values Fitness is the sum of the amplification penalty, the bias penalty, and the two non-linearity penalties.

First, the amplification factor of the circuit is measured using the overall value for gain of the circuit as measured by the slope of the straight line between the output for –10 MV and the output for +10 MV (i.e., between the outputs for the endpoints of the DC sweep). If the amplification factor is less than the target (which is 60 dB for this problem), there is a penalty equal to the shortfall in amplification.

Second, the bias is computed using the DC output associated with a DC input of 0 volts. There is a penalty equal to the bias times a weight. For this problem, a weight of 0.1 is used.

Finally, the linearity is measured by the deviation between the slope of each of two shorter lines and the overall amplification factor of the circuit. The first shorter line segment connects the output value associated with an input of –10 MV and the output value for –5 MV. The second shorter line segment connects the output value for +5 MV and the output for +10 MV. There is a penalty for each of these shorter line segments equal to the absolute value of the difference in slope between the respective shorter line segment and the slope of the straight line between the output for –10 MV and the output for +10 MV.

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

The population size, *M*, was 640,000. This problem was run on a medium-grained parallel Parsytec computer system consisting of 64 80 MHz Power PC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer. The 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 each of the four toroidally adjacent processing nodes. See Andre and Koza 1996 for details.

Architecture-altering operations (Koza 1995) were used.

The other parameters for controlling the runs of genetic programming were the default values specified in Koza 1994 (appendix D).

5.4. Results

About 41% of the circuits of generation 0 cannot be simulated by SPICE; however, the percentage of unsimulatable circuits drops to between 2% and 4% between generations 1 and 10 and never exceeds 8% thereafter.

As the run proceeds from generation to generation, the fitness of the best-of-generation individual tends to improve.

The best circuit (figure 8) from generation 50 has 33 transistors, no diodes, eight capacitors, and five resistors (in addition to the five resistors of the feedback embryo). It achieves a fitness of 971,076.4. No automatically defined functions are present in this particular circuit.

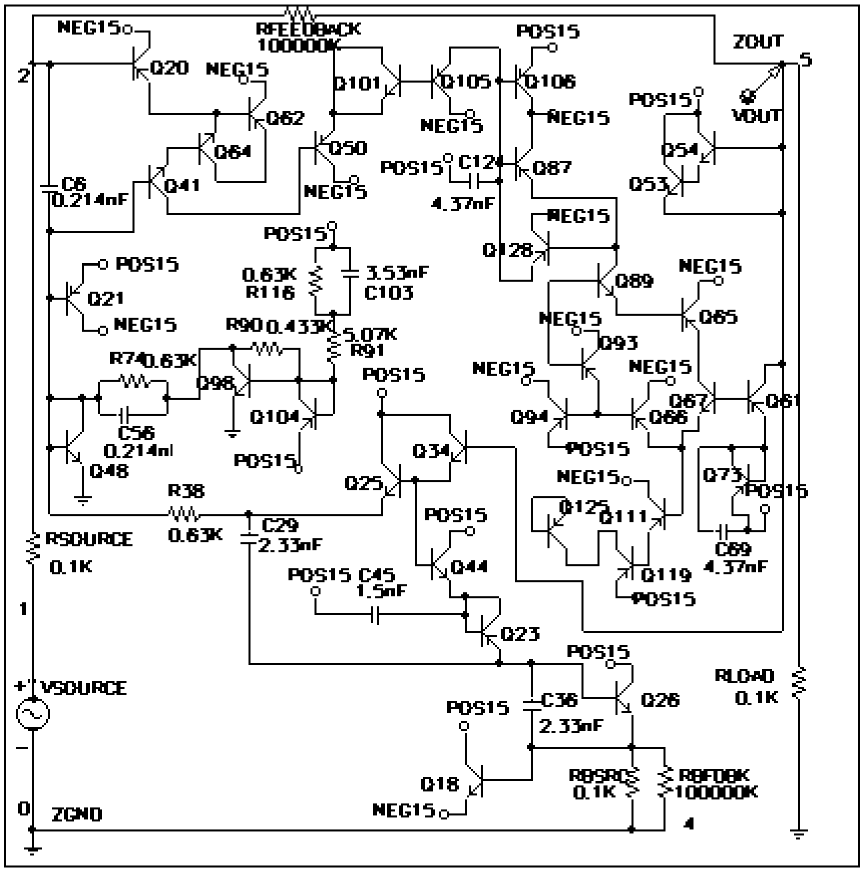

Figure 8 Best circuit from generation 50.

The DC sweep for this best of generation circuit from generation 50 shows that the circuit has an amplification of 89.7 dB (30,545-to-1) and a bias of 9.77 volts.

Based on the time domain behavior of this best circuit from generation 50 and using a 20 microvolt sinusoidal 1,000 Hz signal as input, the amplification is 89.7 dB (30,500-to-1); the bias is 9.76 volts; and the distortion is 6.29%.

Based on the AC sweep for the best circuit of generation 50, the 3 dB bandwidth is 2,300 Hz for a flatband gain is 89.7 dB.

The best circuit (figure 9) from generation 86 achieves a fitness of 938,427.3. The program has two automatically defined functions. ADF0 is called once; ADF1 is not called. The circuit (without ADF0) has 25 transistors, no diodes, two capacitors, and two resistors (in addition to the five resistors of the feedback embryo).

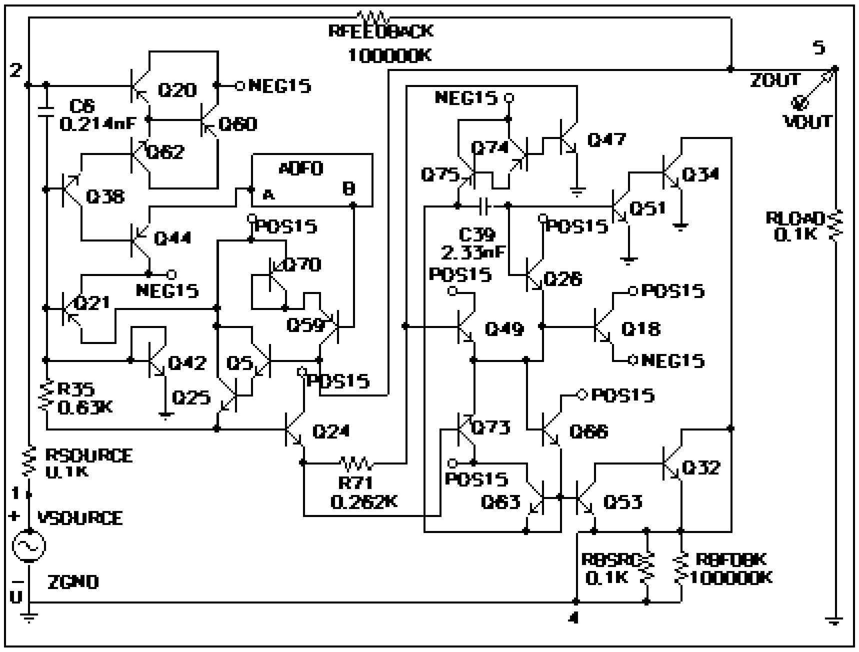

Figure 9 Best circuit from generation 86. Figure 10 shows automatically defined function ADF0 of the best circuit from generation 86 (which has 12 transistors, no diodes, one capacitors, and two resistors).

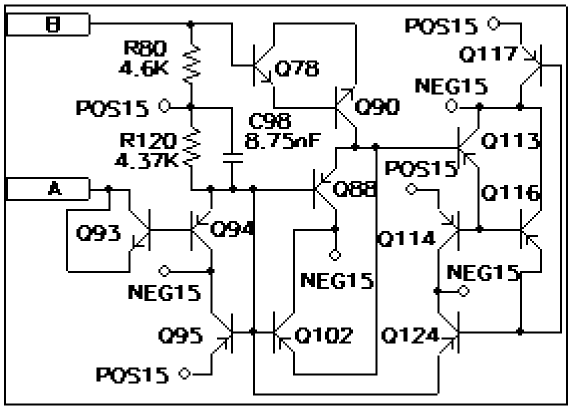

Figure 10 ADF0 for best circuit from generation 86.

The DC sweep for this best of generation circuit from generation 86 shows that the circuit has an amplification of 96 dB.

Based on the time domain behavior of this best circuit from generation 86 and using a 20 microvolt sinusoidal 1,000 Hz signal as input, the amplification is 94.1 dB; the bias is 7.46 volts; and the distortion is 7.07%.

Figure 11 shows the AC sweep for the best circuit of generation 86. The vertical axis ranges from 0 volts to 100 dB. The 3 dB bandwidth is 1078.4 Hz for a flatband gain of 96.3 dB.

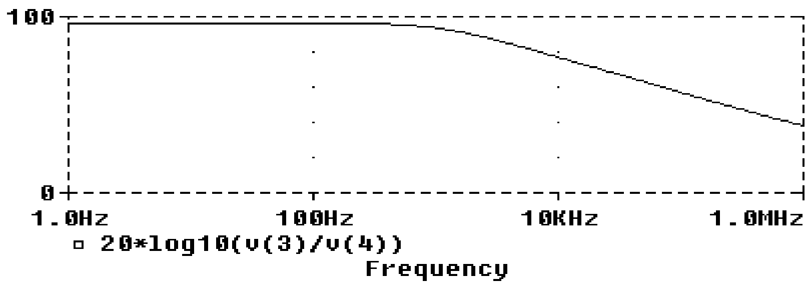

Figure 11 AC sweep for the best circuit from generation 86. We do not claim that the genetically evolved amplifier satisfies all requirements that a human design engineer would want to incorporate into a practical design. We do claim that genetic programming successfully created a 96 dB op amp circuit based on the fitness measure that it was given.

6. Conclusion

We have surveyed four fields in which genetic programming has evolved computer programs that are competitive in performance with human-written programs. We described in detail how genetic programming successfully evolved a 37-transistor amplifier that delivers a gain of 96 dB amplification (based on the DC sweep).

References

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). 1996. *Advances in Genetic Programming 2*. Cambridge: MIT Press.

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.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Gruau, Frederic. 1996. Artificial cellular development in optimization and compilation. In Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag. Pages 48 – 75.

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.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

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. 1995. Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program. *Proceedings of the 14th International Joint Conference on Artificial Intelligence.* San Francisco, CA: Morgan Kaufmann. Pages 734–740.

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., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1997. *Genetic Programming III*. In preparation.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996a. Toward evolution of electronic animals using genetic programming. *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: The MIT Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996b. Four problems for which a computer program evolved by genetic programming is competitive with human performance. *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*. IEEE Press. Pages 1–10.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996c. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In Gero, John S. and Sudweeks, Fay (editors). *Artificial Intelligence in Design '96*. Dordrecht: Kluwer. 151-170.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996d. 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). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT 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.

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. Volume I. Pages 661-664.

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, CA. 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. 13.1.1-13.1.8.

Sanchez, Eduardo and Tomassini, Marco (editors). 1996.*Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag.

Thompson, Adrian. 1996. Silicon evolution. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.