Fourteen Instances where Genetic Programming has Produced Results that are Competitive with Results Produced by Humans

JOHN R. KOZA
Stanford University
Stanford, California 94305
koza@genetic-programming.com
http://www-cs-faculty.stanford.edu/~koza/

FORREST H BENNETT III
Chief Scientist
Genetic Programming Inc.
Los Altos, California 94023
forrest@evolute.com

DAVID ANDRE
University of California
Berkeley, California 94720
dandre@cs.berkeley.edu
www.cs.berkeley.edu/~dandre

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

1. Introduction

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"). Indeed, automatic programming is one of the central problems of computer science.

Paraphrasing Arthur Samuel in 1959 [41], this goal concerns

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

The term "automatic programming" does not have a generally accepted definition. However, a system for automatic programming should reasonably be expected to have the following attributes:

• It starts from a high-level statement of the requirements for a solution to the problem and produces a satisfactory solution to the problem.

• It produces an entity that runs on a computer (i.e., a computer program or other entity that can be easily be translated into a computer program).

• It is problem-independent in the sense that the user does not have to modify the basic operation of the system for each new problem.

• It solves a wide variety of problems from many different fields.

• It has a mechanism for implementing the full range of useful programming constructs (or suitable surrogates of them) including internal memory, data structures, parameterizable subroutines, hierarchical calling of subroutines, iteration, recursion, macros, libraries, multiple inputs, multiple outputs, logical functions, integer functions, floating-point functions, conditional operations, and typing.

• It requires the human user to provide a minimum of information about the nature of the solution in advance.

• It produces the size and shape of the solution to the problem in the sense that it does not require the user to prespecify the exact number of primitive steps that must be performed, the exact sequence of primitive steps, or the arrangement of primitive steps into groups.

• It has a mechanism for automatically organizing sequences of primitive steps into useful groups representing subproblems and of creating a hierarchical organization of the groups of steps.

• It has a mechanism for automatically reusing useful sequences of primitive steps.

• Its scales well to larger versions of the same problem.

• It unmistakably distinguishes between what the user must provide and what the system delivers.

• It operates in a well-defined and automatic way that does not rely on any hidden steps.

• It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers.

The last criterion of the above list is especially important because it reminds us that the ultimate goal of automatic programming is to produce useful programs (not merely programs that solve "proof of principle" or "toy" problems). As Samuel said in 1983 [42],

[T]he aim [is] ... to get machines to exhibit behavior which if done by humans would be assumed to involve the use of intelligence.

An automatically created result can reasonably be viewed as being competitive with one produced by human engineers, designers, mathematicians, or programmers if it satisfies one of the following criteria:

• The result is equal to or better than a recognized human-written algorithm for the same problem.

• The result was patented as an invention in the past, is an improvement over a patented invention, or qualifies today as a patentable new invention.

• The result is equal to or better than a human-produced result accepted and published by a peer-reviewed journal.

• The result was considered an achievement in its field when it was first discovered.

• The result solves a problem that is widely recognized as being difficult by practitioners in the field.

• The result is an improvement over a result produced by an internationally recognized group of experts.

• The result is publishable in its own right (i.e., independent of the fact that the result was mechanically created).

• The result ranks highly or wins a judged competition involving human contestants.

As shown in table 1, this paper refers to 14 instances where genetic programming has produced results that are competitive with those produced by human engineers, designers, mathematicians, or programmers.

The two main points of this paper are as follows:

• **POINT 1:** There are numerous instances (14 of which are shown in table 1) where genetic programming has succeeded in automatically producing computer programs that are competitive with human-produced results.

• **POINT 2:** Genetic programming has all 13 of the above attributes of a system for automatic programming and therefore it can reasonably be asserted that genetic programming *is* automatic programming.

Table 1 Fourteen instances where genetic programming has produced results that are competitive with human-produced results.

	Claimed instance	**Basis for the claim**	**Reference**
1	Automatic creation of the four algorithms for the transmembrane segment identification problem	The result is equal to or better than a recognized human-written algorithm for the same problem.	[18]
2	Automatic creation of a sorting network for seven items with only 16 steps	The result is an improvement over a patented invention.	[29]
3	Automatic synthesis of the recognizable ladder topology of Butterworth or Chebychev lowpass and highpass filters	The result was considered an achievement in its field when it was discovered.	Problems 1 and 2 in this paper and [25]
4	Automatic synthesis of the recognizable "bridged T" topology for filters	The result was considered an achievement in its field when it was discovered.	[25]
5	Automatic synthesis of the recognizable elliptic topology for filters	The result was patented as an invention in the past.	[20]

6	Automatic synthesis of a difficult-to-design asymmetric bandpass filter	The result is equal to or better than a human-produced result accepted and published by a peer-reviewed journal.	[26]
7	Automatic synthesis of a recognizable voltage gain stage and a Darlington emitter-follower section of an ampifier	The result was considered an achievement in its field when it was discovered.	[26]
8	Automatic synthesis of 60 dB and 96 dB amplifiers	The result was considered an achievement in its field when it was discovered.	Problem 6 in this paper and [6], [7], [27]
9	Automatic synthesis of analog computational circuits for squaring, cubing, square root, cube root, and logarithm	The result solves a problem that is widely recognized as being difficult by experts in the field.	Problem 4 in this paper and [31]
10	Automatic synthesis of a real-time analog computational circuit for time-optimal control of a robot	The result solves a problem that is widely recognized as being difficult by practitioners in the field.	Problem 5 in this paper and [30]
11	Automatic synthesis of a voltage reference circuit	The result solves a problem that is widely recognized as being difficult by practitioners in the field.	[28]
12	Automatic creation of a cellular automaton rule for the majority classification problem that is better than the Gacs-Kurdyumov-Levin (GKL) rule and better than all other known rules written by humans over the past 20 years	The result is equal to or better than a recognized human-written algorithm for the same problem.	[3]

13	Automatic creation by genetic programming of motifs that detect the D–E–A-D box family of proteins and the manganese superoxide dismutase family as well, or slightly better than, the human-written motifs found in the PROSITE database	The result is an improvement over a result produced by an internationally recognized group of experts.	[19]
14	Automatic creation of classification programs for identifying extracellular proteins and membrane proteins using programming motifs	The result is equal to or better than a human-produced result accepted and published by a peer-reviewed journal.	[23]

Over the past four decades, the field of artificial intelligence has been dominated by the belief that the goal of getting a computer to solve a problem from a high-level statement of the problem can be achieved by designing a knowledge representation, collecting knowledge from an expert, codifying that knowledge, depositing the codified knowledge into a computer in the form of a knowledge base, and then manipulating the knowledge base using logic. As Lenat [37] stated in 1983,

> All our experiences in AI research have led us to believe that for automatic programming, the answer lies in *knowledge*, in adding a collection of expert rules which guide code synthesis and transformation. (Emphasis in original).

However, the existence of a strongly asserted belief for more than four decades does not preclude the possibility that there may be alternative ways of achieving the goal of automatic programming that do not rely on knowledge or logic. Genetic programming is such an alternative. Genetic programming differs from conventional approaches to artificial intelligence, machine learning, adaptive systems, automated logic, expert systems, and neural networks in the following five fundamental ways:

• Representation: Genetic programming operates directly in the space of computer programs.

• Role of knowledge: Genetic programming operates without a knowledge base.

• Role of logic: None.

• Reliance on transformations of a single point in the search space of the problem to another single point: Genetic programming explicitly exploits a population of candidate approaches (usually contradictory ones) in its search for a solution.

• Mechanism for searching the space of possible solutions: Genetic programming creates programs by means of the Darwinian principle of survival of the fittest, naturally occurring operations such as sexual recombination (crossover), mutation, gene duplication, gene deletion, and the mechanisms of developmental biology.

As to representation, genetic programming operates in the space of possible computer programs. It does not employ the surrogate structures typically used by other techniques of artificial intelligence and machine learning, such as if-then rules, Horn clauses, decision trees, propositional logic, frames, formal grammars, conceptual clusters, concept sets, numerical weight vectors (for neural nets), vectors of numerical coefficients for polynomials (or other fixed expressions), classifier system rules, binary decision diagrams, production rules, expert system rules, or linear chromosome strings (as used in the genetic algorithm). Tellingly, computers are not commonly programmed using any of these surrogate structures. Human programmers do not commonly regard any of them as being useful or suitable for ordinary programming. Our view is that if we are really interested in getting computers to solve problems without explicitly programming them, then the structures that we should be using are *computer programs*. That is, we should be conducting the search for a solution to the problem of automatic programming in the space of computer programs. Computer programs offer the flexibility to perform computations on variables of many different types, perform iterations and recursions, store intermediate results in data structures of various types (named memory, indexed memory, matrices, stacks, lists, rings, queues, relational memory), perform alternative calculations conditionally based on the outcome of other calculations, perform groups of operations in a hierarchical way, and to employ parameterizable, reusable, hierarchically callable subprograms (i.e., subroutines). Subroutines and iterations are particularly important because they involve reusing code and we believe that reusing code is the precondition for scalability in automatic programming as well as other areas of artificial intelligence and machine learning. Of course, once we say that what we really want and need is a computer program, we are immediately faced with the problem of how to locate the desired program in the space of possible programs. That is, we need a way to fruitfully search program space. As will be seen, genetic programming provides a problem-independent way to automatically search the space of programs in order to create a computer program to solve a problem.

Knowledge has no explicit role in genetic programming. There is no knowledge representation; there is no explicit knowledge base in genetic programming; and there is no manipulation of knowledge in genetic programming – by logic or any other means. The practitioners of genetic programming are not against knowledge; they simply say that no one seems to know how, in general, to effectively represent it or to productively manipulate it across a broad range of problems. In any event, an explicit knowledge base is not needed when using genetic programming.

Most research in artificial intelligence is based on methods that are logically sound, correct, consistent, justifiable, deterministic, orderly, parsimonious, and unequivocal. Since computer science is founded on logic, it is almost second nature for computer scientists to unquestioningly assume that every problem-solving technique should be logically sound, correct, consistent, justifiable, deterministic, orderly, parsimonious, and unequivocal. However, logic has no role in evolution in nature and it has no role in genetic programming. Genetic programming is not guided by logic. It operates by cultivating clearly inconsistent and contradictory approaches to solving the problem. When the goal is automatic programming, we believe that the non-logical principles of evolution and natural selection should be used instead of the conventional logic-driven and knowledge-based principles of artificial intelligence. We do not say, "knowledge and logic considered harmful," but simply that neither logic nor any knowledge base are necessary to achieve the goal of automatic programming.

Almost every method in artificial intelligence, machine learning, and neural networks and almost every non-genetic search procedure operates by transforming a single point in the search space of possible solutions to another single point. Because most methods in artificial intelligence, machine learning, and neural networks rely on such point-to-point transformations, they appropriately employ hill-climbing. Genetic programming does not rely on a single point. Instead, genetic programming transforms a population of candidate points into a new (and, on average, better) population of points. Because genetic programming operates on a population, it is free to allocate a certain number of trials to points that, based on the evidence it has at hand, are seemingly inferior. This allocation of a certain measured number of trials to seemingly inferior individuals means that regions of the search space that do not immediately deliver higher payoff are given some attention during the search in the hope that they may lead to regions that do provide ultimately higher payoff.

Concerning its mechanism for searching the space of possible solutions, genetic programming uses the only method that has ever produced highly complex systems and intelligence – the time-tested method of evolution and natural selection. Starting with a primordial ooze of thousands of randomly created computer programs, genetic programming breeds a population of computer programs over many generations by applying the Darwinian principle of survival of the fittest, sexual recombination (crossover), mutation, gene duplication, gene deletion, and the mechanisms of developmental biology.

The basic principles of genetic programming are described in the books *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [14] and *Genetic Programming II: Automatic Discovery of Reusable Programs* [15]. See [35] and [16] for videotape versions of these books. Since l992, over 200 authors have published over 950 papers that have described numerous creative and innovative extensions to the basic idea of genetic programming. The book *Genetic Programming III* [21] is forthcoming.

Since the design process entails creation of a complex structure to satisfy user-defined requirements, the technique of genetic programming will be explained in terms of problems of synthesizing analog electrical circuits in table 1.

The design of analog electrical circuits is particularly challenging because it is generally viewed as requiring human intelligence and because it is a major activity of practicing analog electrical engineers.

The design process for analog circuits begins with a high-level description of the circuit's desired behavior and entails creation of both the topology and the sizing of a satisfactory circuit. The topology comprises the gross number of components in the circuit, the type of each component (e.g., a resistor), and a list of all connections between the components. The sizing involves specifying the values (typically numerical) of each of the circuit's 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 as amenable to automation [40]. Describing "the analog dilemma," Aaserud and Nielsen [1] noted,

> "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."

There has been extensive previous work on the problem of circuit design using simulated annealing, artificial intelligence, and other techniques (as outlined in [28]), including work using genetic algorithms ([36], [9], [43]). However, there has previously been no general automated technique for synthesizing, from scratch, an analog electrical circuit from a high-level statement of the desired behavior of the circuit.

This paper presents a uniform approach to the automatic design of both the topology and sizing of analog electrical circuits. Section 2 presents design problems involving six prototypical analog circuits. Section 3 describes the method. Section 4 details required preparatory steps. Section 5 shows the results for the six problems.

2. Six Problems of Analog Electrical Circuit Design

This paper applies genetic programming to a suite of six problems of analog circuit design. The circuits comprise a variety of types of components, including transistors, diodes, resistors, inductors, and capacitors. The circuits have varying numbers of inputs and outputs.

(1) Design a one-input, one-output lowpass filter composed of capacitors and inductors that passes all frequencies below 1,000 Hz and suppresses all frequencies above 2,000 Hz.

(2) Design a one-input, one-output highpass filter composed of capacitors and inductors that suppresses all frequencies below 1,000 Hz and passes all frequencies above 2,000 Hz.

(3) Design a one-input, one-output tri-state frequency discriminator (source identification) circuit that is composed of resistors, capacitors, and inductors and that produces an output of 1/2 volt and 1 volt for incoming signals whose frequencies are within 10% of 256 Hz and within 10% of 2,560 Hz, respectively, but produces an output of 0 volts otherwise.

(4) Design a one-input, one-output computational circuit that is composed of transistors, diodes, resistors, and capacitors and that produces an output voltage equal to the square root of its input voltage.

(5) Design a two-input, one-output time-optimal robot controller circuit that is composed of the above components and that navigates a constant-speed autonomous mobile robot with nonzero turning radius to an arbitrary destination in minimal time.

(6) Design a one-input, one-output amplifier composed of the above components and that delivers amplification of 60 dB (i.e., 1,000 to 1) with low distortion and low bias.

3. Automated Design by Genetic Programming

The circuits are developed using genetic programming ([14], [35]), an extension of the genetic algorithm [11] in which the population consists of computer programs. Multipart programs consisting of a main program and one or more reusable, parametrized, hierarchically-called subprograms can be evolved using automatically defined functions

([15], [16]). Architecture-altering operations ([17]) automatically determine the number of such subprograms, the number of arguments that each possesses, and the nature of the hierarchical references, if any, among such automatically defined functions. For current research in genetic programming, see [12], [4], [34], [33], [22], and [5].

A computer program is not a design. Genetic programming can be applied to circuits if a mapping is established between the program trees (rooted, point-labeled trees – that is, acyclic graphs – with ordered branches) used in genetic programming and the line-labeled cyclic graphs germane to electrical circuits. The principles of developmental biology, the creative work of Kitano [13] on using genetic algorithms to evolve neural networks, and the innovative work of Gruau [10] on using genetic programming to evolve neural networks provide the motivation for mapping trees into circuits by means of a growth process that begins with an embryo. For circuits, the embryo typically includes fixed wires that connect the inputs and outputs of the particular circuit being designed and certain fixed components (such as source and load resistors). The embryo also contains modifiable wires. The circuit does not produce interesting output until these modifiable wires are modified. A circuit is developed by progressively applying the functions in a program tree to the modifiable wires of the embryo (and, during the developmental process, new modifiable wires and components). See also [8].

The functions in the circuit-constructing program trees are divided into four categories: (1) topology-modifying functions that alter the circuit topology, (2) component-creating functions that insert components into the circuit, (3) arithmetic-performing functions that appear in subtrees as argument(s) to the component-creating functions and specify the numerical value of the component, and (4) automatically defined functions that appear in the function-defining branches and potentially enable certain substructures of the circuit to be reused (with parameterization).

Each branch of the program tree is created in accordance with a constrained syntactic structure. Branches are composed of construction-continuing subtrees that continue the developmental process and arithmetic-performing subtrees that determine the numerical value of components. Topology-modifying functions have one or more construction-continuing subtrees, but no arithmetic-performing subtree. Component-creating functions have one or more construction-continuing subtrees and typically have one arithmetic-performing subtree. This constrained syntactic structure is preserved using structure-preserving crossover with point typing (see [15]).

3.1. The Embryonic Circuit

An electrical circuit is created by executing a circuit-constructing program tree that contains various component-creating and topology-modifying functions. Each tree in the population creates one circuit. The specific embryo used depends on the number of inputs and outputs.

Figure 1 shows a one-input, one-output embryonic circuit in which VSOURCE is the input signal and VOUT is the output signal (the probe point). The circuit is driven by an incoming source VSOURCE. There is a fixed load resistor RLOAD and a fixed source resistor RSOURCE. In addition to the fixed components, there is a modifiable wire Z0 between nodes 2 and 3. All development originates from this modifiable wire.

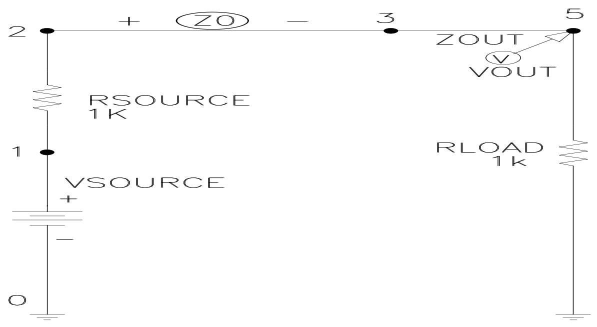

Figure 1 One-input, one-output embryo. **3.2. Component-Creating Functions**

Each program tree contains component-creating functions and topology-modifying functions. The component-creating functions insert a component into the developing circuit and assign component value(s) to the component.

Each component-creating functions has a writing head that points to an associated highlighted component in the developing circuit and modifies that component in a specified manner. The construction-continuing subtree of each component-creating functions points to a successor function or terminal in the circuit-constructing program tree.

The arithmetic-performing subtree of a component-creating functions 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 interpreted on a logarithmic scale 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).

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 (C2, C3, C4, and C5). The circle indicates that Z0 has a writing head (i.e., is the highlighted component and that Z0 is subject to subsequent modification). Figure 3 shows the result of applying the R function to the modifiable wire Z0 of figure 2. The circle indicates that the newly created R1 has a writing head so that R1 remains subject to subsequent modification.

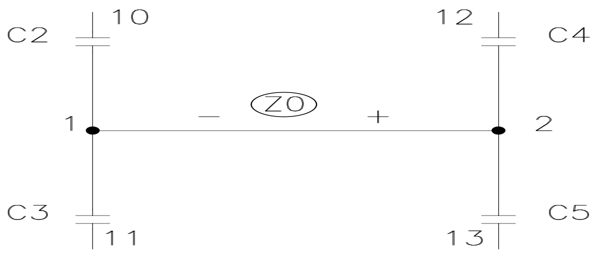

Figure 2 Modifiable wire Z0.

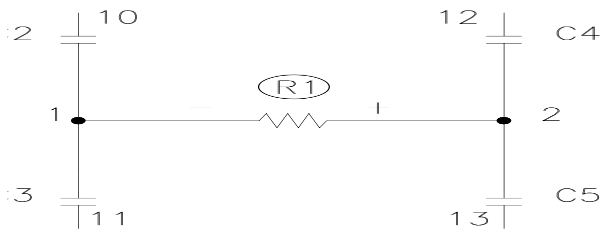

Figure 3 Result of applying the R function.

Similarly, the two-argument capacitor-creating C function causes the highlighted component to be changed into a capacitor whose value in micro Farads is specified by its arithmetic-performing subtrees.

The one-argument Q_D_PNP diode-creating function causes a diode to be inserted in lieu of the highlighted component. This function has only one argument because there is no numerical value associated with a diode and thus no arithmetic-performing subtree. In practice, the diode is implemented here 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 (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 bipolar junction transistor in lieu of the highlighted component and that directly connect the collector or emitter of the newly created transistor to a fixed point of the circuit (the positive power supply, ground, or the negative power supply). For example, the Q_POS_COLL_NPN function inserts a bipolar junction transistor whose collector is connected to the positive power supply.

Each of the functions in the family of six different three-argument transistor-creating Q_3_NPN functions causes an *npn* bipolar junction transistor 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 modifiable wires. There is no writing head on the new transistor, but there is a writing head on each of the three new modifiable wires. There are 12 members (called Q_3_NPN0, ..., Q_3_NPN11) in this family of functions because there are two choices of nodes (1 and 2) to be bifurcated and then there are six ways of attaching the transistor's base, collector, and emitter after the bifurcation. Similarly the family of 12 Q_3_PNP functions causes a *pnp* bipolar junction transistor to be inserted.

3.3. Topology-Modifying Functions

Each topology-modifying function in a program tree points to an associated highlighted component and modifies the topology of the developing circuit.

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

The four-argument PARALLEL0 parallel division function creates a parallel composition consisting of the original highlighted component (with a writing head), a copy of it (with a writing head), two new modifiable wires (each with a writing head), and two new nodes. Figure 4 shows the result of applying PARALLEL0 to the resistor R1 from figure 3.

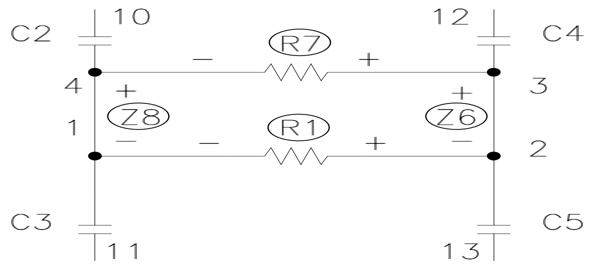

Figure 4 Result of the PARALLEL0 function.The one-argument polarity-reversing FLIP function reverses the polarity of the highlighted component.

There are six three-argument functions (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 then make a connection to ground, positive power supply, or negative power supply, respectively.

There are two three-argument functions (PAIR_CONNECT_0 and PAIR_CONNECT_1) that 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 creates two new wires and one new node, connects the temporary port to the end of one of these new wires, and then removes the temporary port.

The one-argument NOOP 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, 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).

4. Preparatory Steps

Before applying genetic programming to a problem of circuit design, seven major preparatory steps are required: (1) identify the suitable embryonic circuit, (2) determine the architecture of the overall circuit-constructing program trees, (3) identify the terminals of the program trees, (4) identify the primitive functions contained in these program trees, (5) create the fitness measure, (6) choose parameters, and (7) determine the termination criterion and method of result designation.

4.1. Embryonic Circuit

The embryonic circuit used on a particular problem depends on the circuit's number of inputs and outputs.

For example, in the robot controller circuit, the circuit has two inputs, VSOURCE1 and VSOURCE2, not just one. Moreover, each input needs its own separate source resistor (RSOURCE1 and RSOURCE2). Consequently, the embryo has three modifiable wires Z0, Z1, and Z2 in order to provide full connectivity between the two inputs and the one output. All development then originates from these three modifiable wires.

There is often considerable flexibility in choosing the embryonic circuit. For example, an embryo with two modifiable wires (Z0 and Z1) was used for the lowpass and highpass filters. In some problems, such as the amplifier, the embryo contains additional fixed components because of additional problem-specific functionality of the harness (as described in [27] and [28]).

4.2. Program Architecture

Since there is one result-producing branch in the program tree for each modifiable wire in the embryo, the architecture of each circuit-constructing program tree depends on the embryonic circuit. One result-producing branch was used for the frequency discriminator and the computational circuit; two were used for lowpass and highpass filter problems; and three were used for the robot controller and amplifier. The architecture of each circuit-constructing program tree also depends on the use, if any, of automatically defined functions. Automatically defined functions and architecture-altering operations were used in the frequency discriminator, robot controller, and amplifier. For these problems, each program in the initial population of programs had a uniform architecture with no automatically defined functions. In later generations, the number of automatically defined functions, if any, emerged as a consequence of the architecture-altering operations.

4.3. Function and Terminal Sets

The function set for each design problem depended on the type of electrical components that were used to construct the circuit. Inductors and capacitors were used for the lowpass and highpass filter problems. Capacitors, diodes, and transistors were used for the computational circuit, the robot controller, and the amplifier. Resistors (in addition to inductors and capacitors) were used for the frequency discriminator. When transistors were used, functions to provide connectivity to the positive and negative power supplies were also included.

For the computational circuit, the robot controller, and the amplifier, the function set, $\mathcal{F}_{\text{ccs-initial}}$, for each construction-continuing subtree was

$\mathcal{F}_{\text{ccs-initial}}$ = {R, C, SERIES, PARALLEL0, PARALLEL1, FLIP, NOOP, T_GND_0, T_GND_1, T_POS_0, T_POS_1, T_NEG_0, 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 was

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

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

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

where $\leftarrow$ represents floating-point random constants from –1.0 to +1.0.

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

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

The terminal and function sets were identical for all result-producing branches for a particular problem.

For the lowpass filter, highpass filter, and frequency discriminator, there was no need for functions to provide connectivity to the positive and negative power supplies.

For the frequency discriminator, the robot controller, and the amplifier, the architecture-altering operations were used and the set of potential new functions, $\mathcal{F}_{\text{potential}}$, was

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

The set of potential new terminals, $\mathcal{T}_{\text{potential}}$, for the automatically defined functions was

$\mathcal{T}_{\text{potential}} = \{$`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 that

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

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

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

4.4. Fitness Measure

The fitness measure varies for each problem. The high-level statement of desired circuit behavior is translated into a well-defined measurable quantity that can be used by genetic programming to guide the evolutionary process. The evaluation of each individual circuit-constructing program tree in the population begins with its execution. This execution progressively applies the functions in each program tree to an embryonic circuit, thereby creating a fully developed circuit. A netlist is created that identifies each component of the developed circuit, the nodes to which each component is connected, and the value of each component. The netlist becomes the input to the 217,000-line SPICE (Simulation Program with Integrated Circuit Emphasis) simulation program ([38]). SPICE then determines the behavior of the circuit. It was necessary to make considerable modifications in SPICE so that it could run as a submodule within the genetic programming system.

4.4.1. Lowpass Filter

A simple *filter* is a one-input, one-output electronic circuit that receives a signal as its input and passes the frequency components of the incoming signal that lie in a specified range (called the *passband*) while suppressing the frequency components that lie in all other frequency ranges (the *stopband*).

The desired lowpass LC filter should have a passband below 1,000 Hz and a stopband above 2,000 Hz. The circuit is driven by an incoming AC voltage source with a 2 volt amplitude. If the source (internal) resistance RSOURCE and the load resistance RLOAD in the embryonic circuit are each 1 kilo Ohm, the incoming 2 volt signal is divided in half.

The *attenuation* of the filter is defined in terms of the output signal relative to the reference voltage (half of 2 volt here). A *decibel* is a unitless measure of relative voltage that is defined as 20 times the common (base 10) logarithm of the ratio between the voltage at a particular probe point and a reference voltage.

In this problem, a voltage in the passband of exactly 1 volt and a voltage in the stopband of exactly 0 volts is regarded as ideal. The (preferably small) variation within the passband is called the *passband ripple*. Similarly, the incoming signal is never fully reduced to zero in the stopband of an actual filer. The (preferably small) variation within the stopband is called the *stopband ripple*. A voltage in the passband of between 970 millivolts and 1 volt (i.e., a passband ripple of 30 millivolts or less) and a voltage in the stopband of between 0 volts and 1 millivolts (i.e., a stopband ripple of 1 millivolts or less) is regarded as acceptable. Any voltage lower than 970 millivolts in the passband and any voltage above 1 millivolts in the stopband is regarded as unacceptable.

A fifth-order *elliptic* (*Cauer*) *filter* with a modular angle Θ of 30 degrees (i.e., the arcsin of the ratio of the boundaries of the passband and stopband) and a reflection coefficient ρ of 24.3% is required to satisfy these design goals.

Since the high-level statement of behavior for the desired circuit is expressed in terms of frequencies, the voltage VOUT is measured in the frequency domain. SPICE performs an AC small signal analysis and report the circuit's behavior over five decades (between 1 Hz and 100,000 Hz) with each decade being divided into 20 parts (using a logarithmic scale), so that there are a total of 101 fitness cases.

Fitness is measured in terms of the sum over these cases of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit at the probe point VOUT and the target value for voltage. The smaller the value of fitness, the better. A fitness of zero represents an (unattainable) ideal filter.

Specifically, the standardized fitness is

$$F(t) = \sum_{i=0}^{100} \left(W(d(f_i), f_i) d(f_i) \right)$$

where f_i is the frequency of fitness case i; *d(x)* is the absolute value of the difference between the target and observed values at frequency x; and $W(y,x)$ is the weighting for difference y at frequency x.

The fitness measure is designed to not penalize ideal values, to slightly penalize every acceptable deviation, and to heavily penalize every unacceptable deviation. Specifically, the procedure for each of the 61 points in the 3-decade interval between 1 Hz and 1,000 Hz for the intended passband is as follows:

- If the voltage equals the ideal value of 1.0 volt in this interval, the deviation is 0.0.
- If the voltage is between 970 millivolts and 1 volt, the absolute value of the deviation from 1 volt is weighted by a factor of 1.0.
- If the voltage is less than 970 millivolts, the absolute value of the deviation from 1 volt is weighted by a factor of 10.0.

The acceptable and unacceptable deviations for each of the 35 points from 2,000 Hz to 100,000 Hz in the intended stopband are similarly weighed (by 1.0 or 10.0) based on the amount of deviation from the ideal voltage of 0 volts and the acceptable deviation of 1 millivolts.

For each of the five "don't care" points between 1,000 and 2,000 Hz, the deviation is deemed to be zero.

The number of “hits” for this problem (and all other problems herein) is defined as the number of fitness cases for which the voltage is acceptable or ideal or that lie in the "don't care" band (for a filter).

Many of the random initial circuits and many that are created by the crossover and mutation operations in subsequent generations cannot be simulated by SPICE. These circuits receive a high penalty value of fitness (10^8) and become the worst-of-generation programs for each generation.

For details, see Koza, Bennett, Andre, and Keane [25].

4.4.2. Highpass Filter

The fitness cases for the highpass filter are the same 101 points in the five decades of frequency between 1 Hz and 100,000 Hz as for the lowpass filter. The fitness measure is substantially the same as that for the lowpass filter problem above, except that the locations of the passband and stopband are reversed.

4.4.3. Tri-state Frequency Discriminator

Fitness is the sum, over 101 fitness cases, of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit and the target value.

The three points that are closest to the band located within 10% of 256 Hz are 229.1 Hz, 251.2 Hz, and 275.4 Hz. The procedure for each of these three points is as follows: If the voltage equals the ideal value of 1/2 volts in this interval, the deviation is 0.0. If the voltage is more than 240 millivolts from 1/2 volts, the absolute value of the deviation from 1/2 volts is weighted by a factor of 20. If the voltage is more than 240 millivolts of 1/2 volts, the absolute value of the deviation from 1/2 volts is weighted by a factor of 200. This arrangement reflects the fact that the ideal output voltage for this range of frequencies is 1/2 volts, the fact that a 240 millivolts discrepancy is acceptable, and the fact that a larger discrepancy is not acceptable.

Similar weighting was used for the three points (2,291 Hz, 2,512 Hz, and 2,754 Hz) that are closest to the band located within 10% of 2,560 ,Hz.

The procedure for each of the remaining 95 points is as follows: If the voltage equals the ideal value of 0 volts, the deviation is 0.0. If the voltage is within 240 millivolts of 0 volts, the absolute value of the deviation from 0 volts is weighted by a factor of 1.0. If the voltage is more than 240 millivolts from 0 volts, the absolute value of the deviation from 0 volts is weighted by a factor of 10. For details, see [32].

4.4.4. Computational Circuit

SPICE is called to perform a DC sweep analysis at 21 equidistant voltages between –250 millivolts and +250 millivolts. Fitness is the sum, over these 21 fitness cases, of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit and the target value for voltage. For details, see 31].

4.4.5. Time-Optimal Robot Controller Circuit

The goal in this time-optimal navigation problem is to find a strategy for continuously controlling the movement of a constant-speed moving object with a nonzero turning radius to an arbitrary destination point. If the robot has a nonzero turning radius, this problem cannot be solved merely by executing a hill-climbing action that greedily improves the distance between the robot and the destination at every intermediate point along the robot's trajectory. Instead, momentarily disadvantageous actions must be taken in the short term in order to achieve the long-term objective.

An evolved computer program could be installed in an actual robot by writing a computer program for an embeded digital controller residing in the robot. When the robot with this embeded digital computer is operating, the robot's current location would be sensed in analog form. This analog input would then be converted into digital form using an analog-to-digital converter. Then, the embeded digital processor would perform a digital calculation consisting of a sequence of arithmetic and conditional operations on the digital data. Finally, the digital output would be passed to a digital-to-analog converter that would convert the program's digital output into an analog signal. The resulting analog signal would, in turn, cause the robot to move in the specified direction.

Notice that the above approach has two major phases. First, the time-optimal computer program for controlling the robot must be created (say, by genetic programming). Secondly, the computer program must be installed in a digital computer resident in the robot. This section illustrates a more holistic approach to this problem in which there is one phase, instead of two. The goal here to evolve the design for a real-time analog circuit for directly controlling a robot in the required time-optimal way. There will be no intermediate computer program consisting of addition, subtraction, multiplication, division, or conditional branching operatons; and there will be no digital prcoessor resident in the robot. More specifically, the goal here is to evolve the design for a real-time analog electrical circuit for controlling the trajectory of a robot with nonzero turning radius (moving at constant speed) such that the robot moves to an arbitrary destination point in minimal time.

The fitness of a robot controller was evaluated using 72 randomly chosen fitness cases each representing a different target point. Fitness is the sum, over the 72 fitness cases, of the travel times. If the robot came within a capture radius of 0.28 meters of its target point before the end of the 80 time steps allowed for a particular fitness case, the contribution to fitness for that fitness case was the actual time. However, if the robot failed to come within the capture radius during the 80 time steps, the contribution to fitness was 0.160 hours (i.e., double the worst possible time).

SPICE performs a nested DC sweep, which provides a way to simulate the DC behavior of a circuit with two inputs. It resembles a nested pair of `FOR` loops in a computer program in that both of the loops have a starting value for the voltage, an increment, and an ending value for the voltage. For each voltage value in the outer loop, the inner loop simulates the behavior of the circuit by stepping through its range of voltages. Specifically, the starting value for voltage is –4 volt, the step size is 0.2 volt, and the ending value is +4 volt. These values correspond to the dimensions of the robot's world of 64 square meters extending 4 meters in each of the four directions from the origin of a coordinate system (i.e., 1 volt equals 1 meter). For details, see [30].

4.4.6. 60 dB Amplifier

SPICE was requested to perform a DC sweep analysis to determine the circuit's response for several different DC input voltages. An ideal inverting amplifier circuit would receive the 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, the output signal is not perfectly centered on 0 volts(i.e., it is biased), or the DC response is not linear. Fitness is calculated by summing an amplification penalty, a bias penalty, and two non-linearity penalties – each derived from these five DC outputs. For details, see [7].

4.5. Control Parameters

The population size, *M*, was 640,000 for all problems. Other parameters were substantially the same for each of the six problems and can be found in the references cited above.

4.6. Implementation on Parallel Computer

Each problem was run on a medium-grained parallel Parsytec computer system [2] consisting of 64 80-MHz PowerPC 601 processors arranged in an 8 by 8 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 (semi-isolated subpopulations). 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 adjacent processing nodes.

5. Results

In all six problems, fitness was observed to improve over successive generations. A large majority of the randomly created initial circuits of generation 0 were not able to be simulated by SPICE; however, most were simulatable after only a few generations. Satisfactory results were generated in every case on the first or second trial. When two runs were required, the first produced an almost satisfactory result. This rate of success suggests that the capabilities of the approach and current computing system have not been fully exploited.

5.1. Lowpass Filter

Many of the runs produced lowpass filters having a topology similar to that employed by human engineers. For example, in generation 32 of one run, a circuit (figure 5) was evolved with a near-zero fitness of 0.00781. The circuit was 100% compliant with the design requirements in that it scored 101 hits (out of 101). After the evolutionary run, this circuit (and all evolved circuits herein) were simulated anew using the commercially available MicroSim circuit simulator to verify performance. As can be seen, inductors appear in series horizontally across the top of the figure, while capacitors appear vertically as shunts to ground. This circuit had the recognizable ladder topology (46) of a Butterworth or Chebychev lowpass filter.

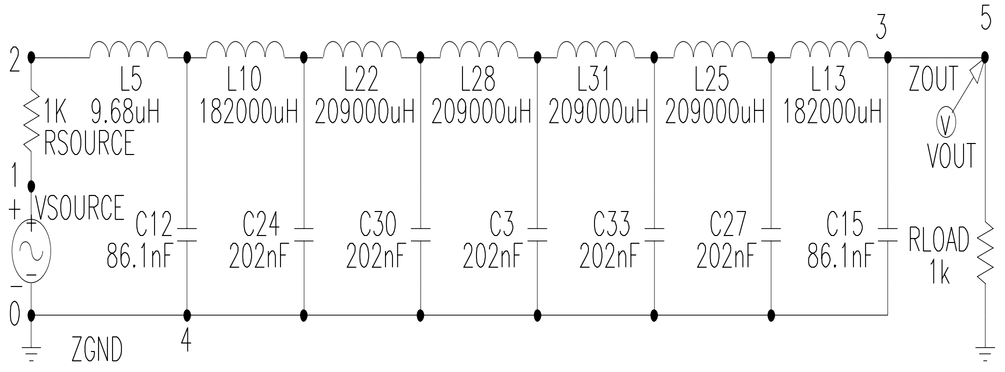

Figure 5 Evolved 7-rung ladder lowpass filter. Figure 6 shows the behavior in the frequency domain of this evolved lowpass filter. As can be seen, the evolved circuit delivers about 1 volt for all frequencies up to 1,000 Hz and about 0 volts for all frequencies above 2,000 Hz.

In another run, a 100% compliant recognizable "bridged T" arrangement was evolved. In yet another run using automatically defined functions, a 100% compliant circuit emerged with the recognizable elliptic topology that was invented and patented by Cauer. When invented. the Cauer filter was a significant advance (both theoretically and commercially) over the Butterworth and Chebychev filters.

Thus, genetic programming rediscovered the ladder topology of the Butterworth and Chebychev filters, the "bridged T" topology, and the elliptic topology.

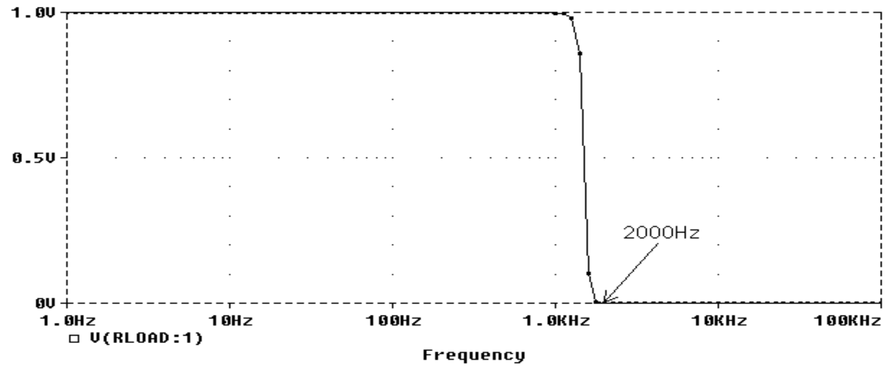

Figure 6 Frequency domain behavior of genetically evolved 7-rung ladder lowpass filter.

It is important to note that when we performed the preparatory steps for applying genetic programming to the problem of synthesizing a lowpass filter, we did not employ any significant domain knowledge from the field of electrical engineering. We did not incorporate knowledge of Kirchhoff's laws, integro-differential equations, Laplace transforms, poles, zeroes, or the other mathematical techniques and insights about filters that are known to electrical engineers who design analog filters. In spite of this absence of explicit domain knowledge, genetic programming evolved a 100% compliant circuit for the problem of designing an LC lowpass filter that embodied the ladder topology that is well known in the field of electrical engineering.

The reinvention by genetic programming of the recognizable ladder topology of Butterworth or Chebychev lowpass filters is an instance where genetic programming has produced a result that is competitive with those created by inventive and knowledgeable humans. It satisfies Arthur Samuel's criterion [42] for artificial intelligence and machine learning, namely

> [T]he aim [is] ... to get machines to exhibit behavior which if done by humans would be assumed to involve the use of intelligence.

5.2. Highpass Filter

In generation 27 of one run, a 100% compliant circuit (figure 7) was evolved with a near-zero fitness of 0.213. This circuit has four capacitors and five inductors (in addition to the fixed components of the embryo). Capacitors appear in series horizontally across the top of the figure, while inductors appear vertically as shunts to ground.

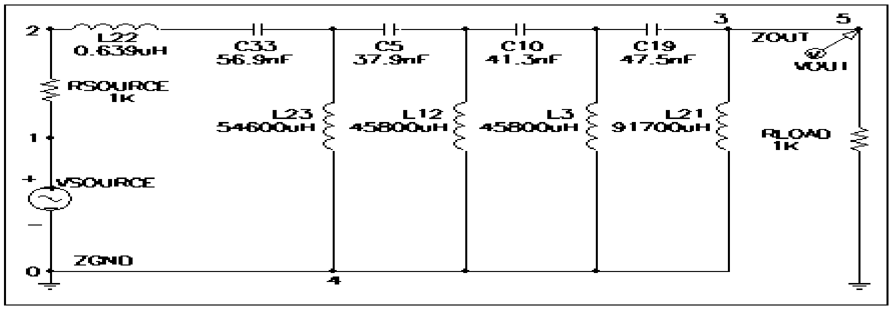

Figure 7 Evolved four-rung ladder highpass filter. Figure 8 shows the behavior in the frequency domain of this evolved highpass filter. As desired, the evolved highpass delivers about 0 volts for all frequencies up to 1,000 Hz and about 1 volt for all frequencies above 2,000 Hz.

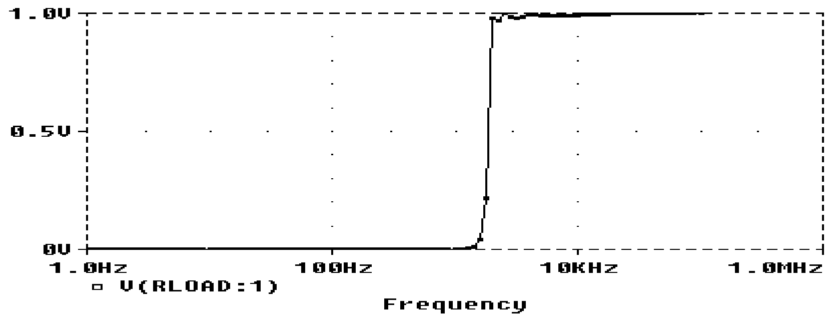

Figure 8 Frequency domain behavior of evolved four-rung ladder highpass filter. The reversal of roles for the capacitors and inductors in lowpass and highpass ladder filters is well known to electrical engineers and can be elegantly explained based on the duality of the single terms (derivatives versus integrals) in the integro-differential equations that represent the voltages and currents of the inductors and capacitors in the loops and nodes of a circuit. However, genetic programming was not given any domain knowledge concerning the duality of anything. In fact, the fitness measure was the only difference between the preparatory steps for the highpass and lowpass filters. Thus, in spite of the absence of explicit domain knowledge about duality or electrical engineering, genetic programming evolved a 100% compliant highpass filter embodying the appropriate highpass ladder topology. Using the fitness measure appropriate for highpass filters, genetic programming searched the same space (i.e., the space of circuit-constructing program trees composed of the same component-creating functions and the same topology-modifying functions) and discovered circuit-constructing program tree that yielded a 100%-complaint highpass filter.

The rediscovery by genetic programming of the reversal of roles of capacitors and inductors in lowpass and filters is an instance where genetic programming has produced a result that is competitive with those created by inventive and knowledgeable humans. This result (and all of the succeeding results in this paper) satisfies Arthur Samuel's criterion [42] for success in artificial intelligence and machine learning.

5.3. Tri-state Frequency Discriminator

The evolved three-way tri-state frequency discriminator circuit from generation 106 scores 101 hits (out of 101). Figure 9 shows this circuit (after expansion of its automatically defined functions). The circuit produces the desired outputs of 1 volt and 1/2 volts (each within the allowable tolerance) for the two specified bands of frequencies and the desired near-zero signal for all other frequencies.

5.4. Computational Circuit

The genetically evolved computational circuit for the square root from generation 60 (figure 10), achieves a fitness of 1.68, and has 36 transistors, two diodes, no capacitors, and 12 resistors (in addition to the source and load resistors in the embryo). The output voltages produced by this best-of-run circuit are almost exactly the required values.

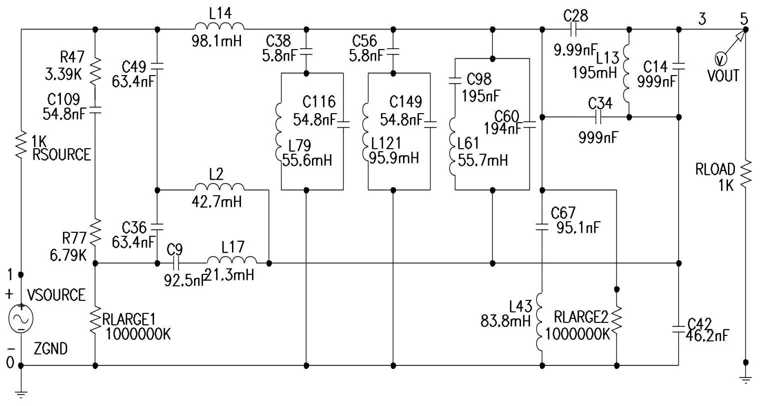

Figure 9 Evolved frequency discriminator.

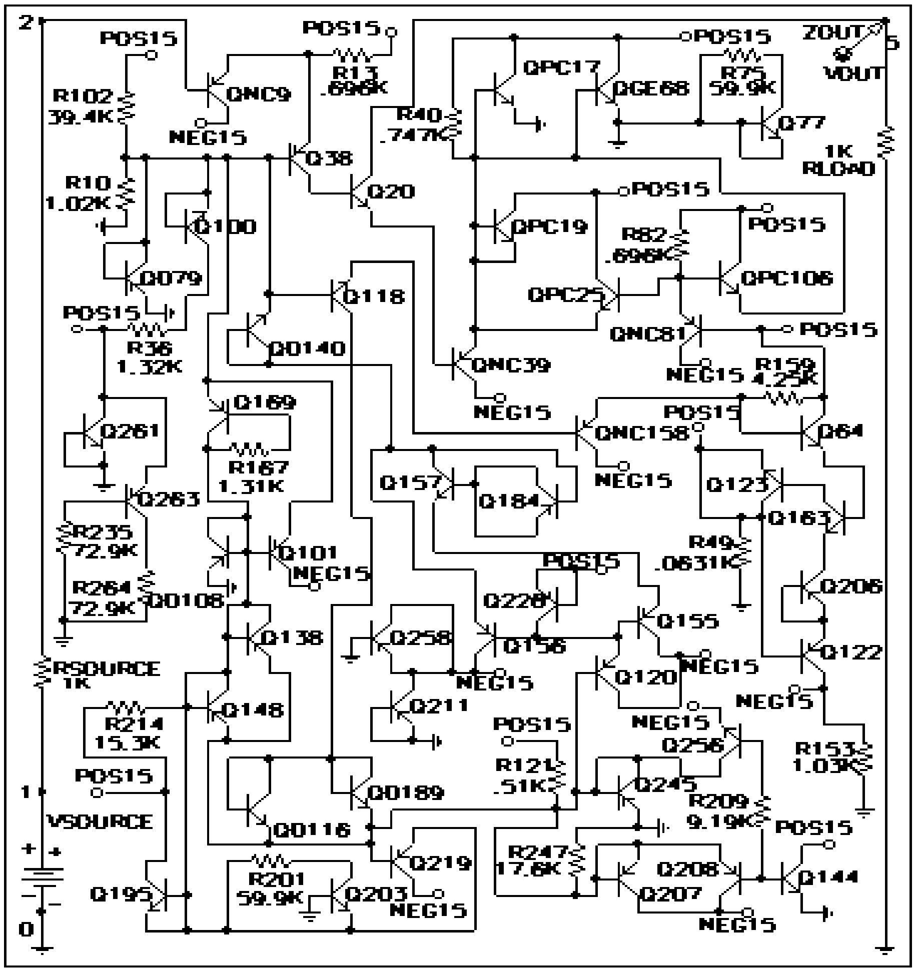
Figure 10 Evolved square root circuit.

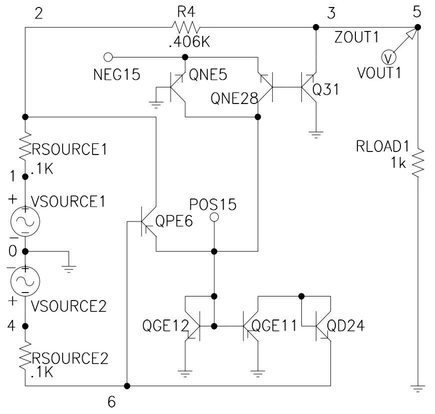

Figure 11 Best circuit of generation 0.

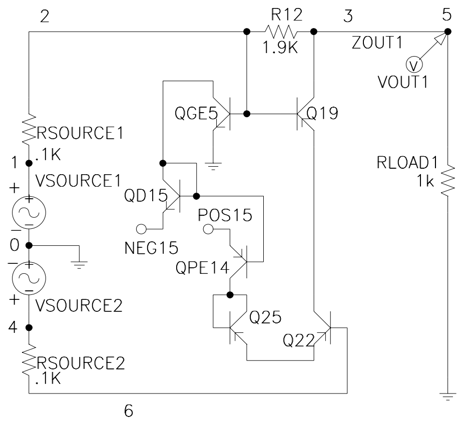

Figure 12 Best circuit of generation 8.

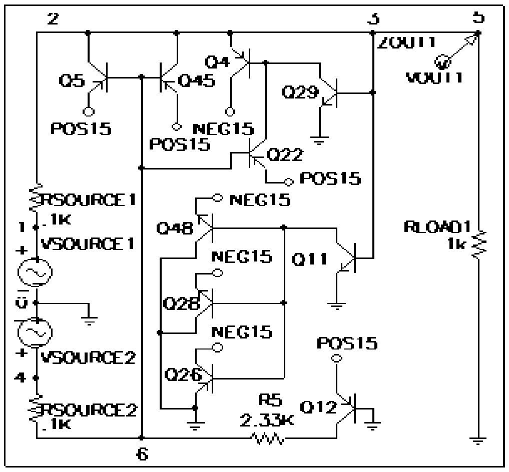

Figure 13 Best circuit of generation 17.

5.5. Robot Controller Circuit

The best circuit from generation 0 (figure 11) scores 61 hits (out of 72) and achieves a fitness of 3.005 hours and has seven transistors, one diode, and one resistor (not counting the three resistors of the harness and those contained in the power supplies). Its circuit-constructing program tree has four, 48, and 11 points, respectively, in its three result-producing branches.

The best circuit from generation 8 (figure 12) scores 67 hits and achieves a fitness of 2.57 hours and has seven transistors, one diode, and one resistor.

The best circuit (figure 13) from generation 17 scores 69 hits and achieves a fitness of 2.06 hours and has 12 transistors, no diodes, and one resistor. .

The best circuit of generation 20 scores 72 hits (out of 72) and achieves a fitness of 1.602 hours. This is the first robot controller circuit that reaches all 72 destinations within the alloted time.

The best-of-run time-optimal robot controller circuit (figure 14) appeared in generation 31, scores 72 hits, and achieves a near-optimal fitness of 1.541 hours. In comparison, the optimal value of fitness for this problem is known to be 1.518 hours. This best-of-run circuit has 10 transistors and 4 resistors.

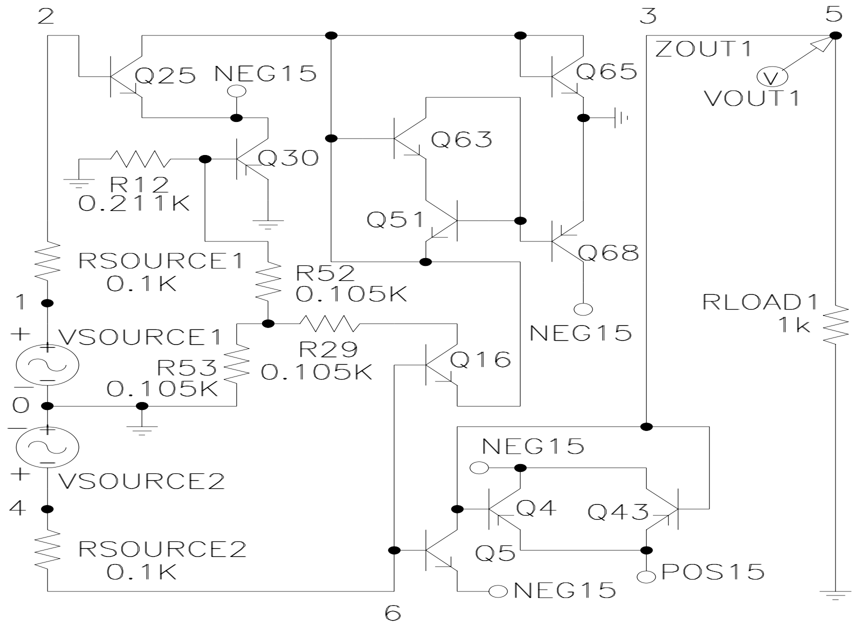

Figure 14 Evolved real-time robot controller circuit.

5.6. 60 dB Amplifier

The best circuit from generation 109 (figure 15) achieves a fitness of 0.178. Based on a DC sweep, the amplification is 60 dB here (i.e., 1,000-to-1 ratio) and the bias is 0.2 volt. Based on a transient analysis at 1,000 Hz, the amplification is 59.7 dB; the bias is 0.18 volts; and the distortion is very low (0.17%). Based on an AC sweep, the amplification at 1,000 Hz is 59.7 dB; the flatband gain is 60 dB; and the 3 dB bandwidth is 79, 333 Hz. Thus, a high-gain amplifier with low distortion and acceptable bias has been evolved.

5.7. Other Circuits

Numerous other circuits have been similarly designed, including asymmetric bandpass filters [26], crossover filters [24], double passband filters [20], amplifiers with high power supply rejection ratio [6], other amplifiers [27], a temperature-sensing circuit, and a voltage reference circuit [28].

5.8. Another Robotics-Related Genetically Evolved Program

There are other instances where genetic programming has created a computer program that is competitive with humans. For example, at the RoboCup97 competition at the International Joint Conference on Artificial Intelligence in Nagoya, Japan in August 1997, Sean Luke's entry [39] was a soccer-playing program that was evolved by genetic programming. The other entries in the competition were computer programs written by teams of humans. Luke's entry came in third place in this international competition and received the Science award for the competition.

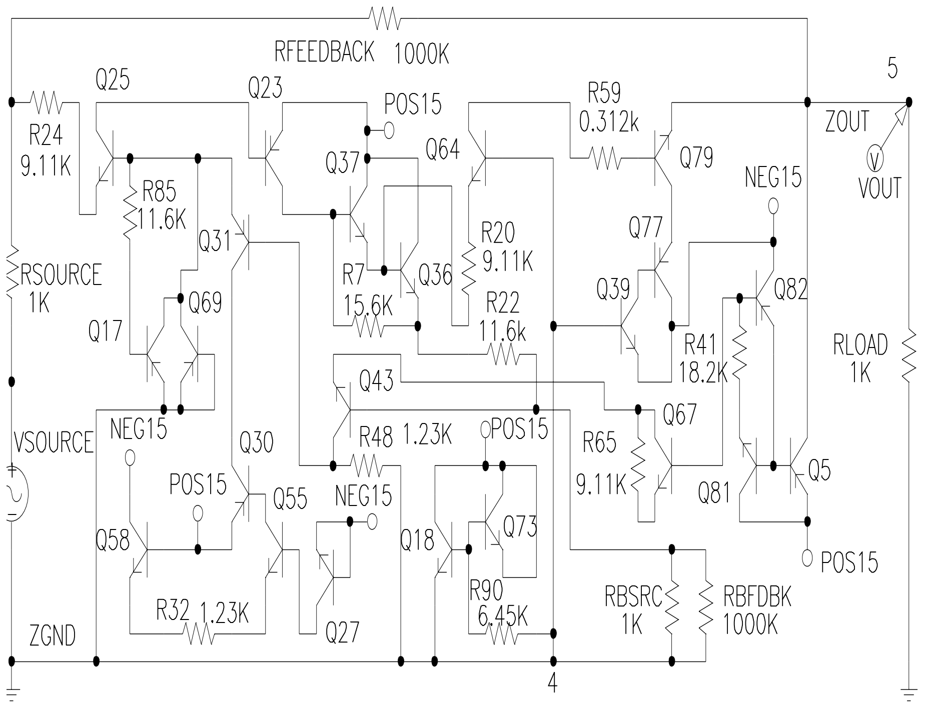

Figure 15 Evolved amplifier.

6. Conclusion

Genetic programming evolved the topology and sizing of six different prototypical analog electrical circuits, including a lowpass filter, a highpass filter, a tri-state frequency discriminator circuit, a 60 dB amplifier, a computational circuit for the square root, and a time-optimal robot controller circuit. All six of these genetically evolved circuits constitute instances of an evolutionary computation technique solving a problem that is usually thought to require human intelligence.

References

[1] 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.

[2] 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.

[3] 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). 1996. *Genetic Programming 1996:*

Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University. Cambridge, MA: MIT Press. Pages 3–11.

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

[5] Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1997. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

[6] Bennett III, Forrest H. 1997. Programming computers by means of natural selection: Applicaton to analog circuit synthesis. In Hara, F. and Yoshida, K. (editors). *Proceedings of International Symposium on System Life, July 21 – 22, 1997, Tokyo International Forum*. Tokyo: The Japan Society of Mechanical Engineers. Pages 41 – 50.

[7] Bennett III, Forrest H, Koza, John R., Andre, David, and Keane, Martin A. 1996. Evolution of a 60 Decibel op amp using genetic programming. In Higuchi, Tetsuya, Iwata, Masaya, and Lui, Weixin (editors). *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*. Lecture Notes in Computer Science, Volume 1259. Berlin: Springer-Verlag. Pages 455-469.

[8] Brave, Scott. 1996. Evolving deterministic finite automata using cellular encoding. 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. Pages 39–44.

[9] Grimbleby, J. B. 1995. Automatic analogue network synthesis using genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications.* London: Institution of Electrical Engineers. Pages 53–58.

[10] 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.

[11] Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

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

[13] Kitano, Hiroaki. 1990. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*. 4(1990) 461–476.

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

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

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

[17] Koza, John R. 1995. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, John R., Reynolds, Robert G., and Fogel, David B. (editors). 1995. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. Pages 695–717.

[18]Koza, John R. and Andre, David. 1996b. Evolution of iteration in genetic programming. In Fogel, Lawrence J., Angeline, Peter J. and Baeck, T. *Evolutionary*

Programming V: *Proceedings of the Fifth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. Pages 469 – 478.

[19] Koza, John R. and Andre, David. 1998. Automatic discovery of protein motifs using genetic programming. In Yao, Xin (editor). 1997. *Evolutionary Computation: Theory and Applications*. Singapore: World Scientific. In Press.

[20] 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*. Cambridge, MA: The MIT Press.

[21] Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1998. *Genetic Programming III*. San Francisco, CA: Morgan Kaufmann. Forthcoming.

[22] Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoym Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). 1998. *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of Wisconsin, Madison, Wisconsin*. San Francisco, CA: Morgan Kaufmann.

[23] Koza, John R., Bennett III, Forrest H, and Andre, David. 1998. Using programmatic motifs and genetic programming to classify protein sequences as to extracellular and membrane cellular location. *Evolutionary Programming VII. 67h International Conference, EP98, San Diego, USA, march 1998 Proceedings*. Lecture Notes in Computer Science, Berlin: Springer-Verlag. In Press.

[24] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996a. 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. 1–10.

[25] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996b. 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. Pages 151-170.

[26] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996c. 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*. Cambridge, MA: The MIT Press.

[27] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1997. Evolution using genetic programming of a low-distortion 96 Decibel operational amplifier. *Proceedings of the 1997 ACM Symposium on Applied Computing, San Jose, California, February 28 – March 2, 1997*. New York: Association for Computing Machinery. Pages 207 - 216.

[28] Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A, and Dunlap, Frank. 1997. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*. 1(2). Pages 109 – 128.

[29] Koza, John R., Bennett III, Forrest H, Hutchings, Jeffrey L., Bade, Stephen L., Keane, Martin A., and Andre, David. 1998. Evolving computer programs using rapidly

reconfigurable field-programmable gate arrays and genetic programming. *Proceedings of the ACM Sixth International Symposium on Field Programmable Gate Arrays*. New York, NY: ACM Press. Pages 209 – 219.

[30] Koza, John R., Bennett III, Forrest H, Keane, Martin A., and Andre, David. 1997. Automatic programming of a time-optimal robot controller and an analog electrical circuit to implement the robot controller by means of genetic programming. *Proceedings of 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*. Los Alamitos, CA; Computer Society Press. Pages 340 – 346.

[31] Koza, John R., Bennett III, Forrest H, Lohn, Jason, Dunlap, Frank, Andre, David, and Keane, Martin A. 1997. Automated synthesis of computational circuits using genetic programming. *Proceedings of the 1997 IEEE Conference on Evolutionary Computation.* Piscataway, NJ: IEEE Press. 447–452.

[32] Koza, John R., Bennett III, Forrest H, Lohn, Jason, Dunlap, Frank, Andre, David, and Keane, Martin A. 1997b. Use of architecture-altering operations to dynamically adapt a three-way analog source identification circuit to accommodate a new source. In Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. San Francisco, CA: Morgan Kaufmann. 213 – 221.

[33] Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. San Francisco, CA: Morgan Kaufmann.

[34] Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

[35] Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

[36] Kruiskamp Marinum Wilhelmus 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.

[37] Lenat, Douglas B. l983. The role of heuristics in learning by discovery: Three case studies. In Michalski, Ryszard S., Carbonell, Jaime G., and Mitchell, Tom M. (editors) *Machine Learning: An Artificial Intelligence Approach, Volume I*. Los Altos, CA: Morgan Kaufmann. Pages 243-306.

[38] 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.

[39] Luke, Sean. Genetic programming and coevolution produced competitive soccer fotbot teams for RoboCup97. In Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoym Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of Wisconsin, Madison, Wisconsin*. San Francisco, CA: Morgan Kaufmann.

[40] 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.

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

[42] Samuel, Arthur L. 1983. AI: Where it has been and where it is going. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann. Pages 1152 – 1157.

[43] 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*. Cambridge, MA: MIT Press.

Version 1 - Submitted March 20, 1998 for International Symposium on Evolutionary Robotics (ISER-98) Conference in Tokyo on April 8 - 9, 1998.

Fourteen Instances where Genetic Programming has Produced Results that are Competitive with Results Produced by Humans

JOHN R. KOZA
Stanford University
Stanford, California 94305
koza@genetic-programming.com
http://www-cs-faculty.stanford.edu/~koza/

FORREST H BENNETT III
Chief Scientist
Genetic Programming Inc.
Los Altos, California 94023
forrest@evolute.com

DAVID ANDRE
University of California
Berkeley, California 94720
dandre@cs.berkeley.edu
www.cs.berkeley.edu/~dandre

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