Automatic Design of Analog Electrical Circuits using Genetic Programming

John R. Koza
Computer Science Dept.
Stanford University
Stanford, California
94305-9020
koza@cs.stanford.edu
http://www-cs-faculty.stanford.edu/~koza/

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
forrest@evolute.com

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

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

ABSTRACT

The design (synthesis) of analog electrical circuits entails the creation of both the topology and sizing (numerical values) of all of the circuit's components. There has previously been no general automated technique for automatically designing an analog electrical circuit from a high-level statement of the circuit's desired behavior.

This chapter introduces genetic programming and shows how it can be used to automate the design of both the topology and sizing of a suite of five prototypical analog circuits, including a lowpass 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.

The problem-specific information required for each of the eight problems is minimal and consists primarily of the number of inputs and outputs of the desired circuit, the types of available components, and a fitness measure that restates the high-level statement of the circuit's desired behavior as a measurable mathematical quantity.

All five of these genetically evolved circuits constitute instances of an evolutionary computation technique solving a problem that is usually thought to require human intelligence.

1. Introduction

The design process entails creation of a complex structure to satisfy user-defined requirements. 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 component.

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 [Rutenbar 1993]. Describing "the analog dilemma," Aaserud and Nielsen [1995] 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 Koza, Bennett, Andre, Keane, and Dunlap 1997, including work using genetic algorithms [Kruiskamp and Leenaerts 1995; Grimbleby 1995; Thompson 1996]. However, there has previously been no general automated technique for synthesizing an analog electrical circuit from a high-level statement of the desired behavior of the circuit.

Once the user has specified the high-level design goals for an analog circuit, it would be ideal if an automated design process could create *both* the topology and the sizing of a circuit that satisfies the design goals. That is, it would be ideal to have an automated "What You Want Is What You Get" ("WYWIWYG" – pronounced "wow-eee-wig") process for analog circuit design.

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 five prototypical analog circuits. Section 3 describes genetic programming. Section 4 describes the method by which genetic programming is applied to the problem of designing analog electrical circuits.. Section 5 details required preparatory steps. Section 6 shows the results for the five problems. Section 7 cites other circuits that have been designed by genetic programming.

2. Five Problems of Analog Design

This paper applies genetic programming to a suite of five 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 lowpass filter having a one-input, one-output composed of capacitors and inductors and that passes all frequencies below 1,000 Hz and suppresses all frequencies above 2,000 Hz.

(2) Design a tri-state frequency discriminator (source identification) circuit having one input and one output 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.

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

(4) Design a time-optimal robot controller circuit having two inputs and one output 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.

(5) Design an 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. Genetic Programming

Genetic programming is a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems. Genetic programming is based on the Darwinian principle of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as *crossover* (*sexual recombination*) and *mutation*.

John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975) described how an analog of the evolutionary process can be applied to solving mathematical problems and engineering optimization problems using what is now called the *genetic algorithm* (GA). The genetic algorithm attempts to find a good (or best) solution to the problem by genetically breeding a population of individuals over a series of generations. In the genetic algorithm, each *individual* in the population represents a candidate solution to the given problem. The *genetic algorithm* (GA) transforms a *population* (set) of individuals, each with an associated *fitness* value, into a new *generation* of the population using reproduction, crossover, and mutation.

Books on genetic algorithms that include those that survey the entire field, such as Goldberg (1989), Michalewicz (1992), and Mitchell (1996) as well as others that specialize in particular areas, such as the application of genetic algorithms to robotics (Davidor (1990), financial applications (Bauer 1994), image segmentation (Bhanu and Lee 1994), pattern recognition (Pal and Wang 1996), parallelization (Stender 1993), and simulation an modeling (Stender, Hillebrand, and Kingdon 1994), control and signal processing (Man, Tang, Kwong, and Halang 1997), and engineering design (Gen and Cheng 1997).

Genetic programming addresses one of the central goals of computer science, namely automatic programming. The goal of automatic programming is to create, in an automated way, a computer program that enables a computer to solve a problem. Paraphrasing Arthur Samuel (1959), the goal of automatic programming concerns,

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

In genetic programming, the genetic algorithm operates on a population of computer programs of varying sizes and shapes (Koza 1992). Genetic programming starts with a primordial ooze of thousands or millions 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 programmatic 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.

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

(1) Randomly create an initial population of individual computer programs.

(2) Iteratively perform the following substeps (called a *generation*) on the population of programs until the termination criterion has been satisfied:

  (a) Assign a fitness value to each individual program in the population using the fitness measure.

  (b) Create a new population of individual programs by applying the following three genetic operations. The genetic operations are applied to one or two individuals in the population selected with a probability based on fitness (with reselection allowed).

    (i) Reproduce an existing individual by copying it into the new population.

    (ii) Create two new individual programs from two existing parental individuals by genetically recombining subtrees from each program using the crossover operation at randomly chosen crossover points in the parental individuals.

    (iii) Create a new individual from an existing parental individual by randomly mutating one randomly chosen subtree of the parental individual.

(3) Designate the individual computer program that is identified by the method of result designation (e.g., the *best-so-far* individual) as the result of the run of genetic programming. This result may represent a solution (or an approximate solution) to the problem.

Genetic programming has been applied to numerous problems in fields such as system identification, control, classification, design, optimization, and automatic programming. Between 1992 and 1997, over 800 papers have been published on genetic programming.

Multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms (called *automatically defined functions*) may also be evolved (Koza 1994a, 1994b). An *automatically defined function* (*ADF*) is a function (i.e., subroutine, subprogram, `DEFUN`, procedure) that is dynamically evolved during a run of genetic programming and which may be called by a calling program (or subprogram) that is concurrently being evolved. When automatically defined functions are being used, a program in the population consists of a hierarchy of one (or more) *reusable* function-defining branches (i.e., automatically defined functions) along with a main result-producing branch. Typically, the automatically defined functions possess one or more dummy arguments (formal parameters) and are reused with different instantiations of these dummy arguments. During a run, genetic programming evolves different subprograms in the function-defining branches of the overall program, different main programs in the result-producing branch, different instantiations of the dummy arguments of the automatically defined functions in the function-defining branches, and different hierarchical references between the branches.

Architecture-altering operations enhance genetic programming with automatically defined functions by providing a way to automatically determine the number of such subprograms, the number of arguments that each subprogram possesses, and the nature of the hierarchical references, if any, among such subprograms (Koza 1995). These operations include branch duplication, argument duplication, branch creation, argument creation, branch deletion, and argument deletion. The architecture-altering operations are motivated by the naturally occurring mechanism of gene duplication that creates new proteins (and hence new structures and new behaviors in living things) (Ohno 1970).

Recent research on genetic programming is described in books (Banzhaf, Nordin, Keller, and Francone 1997), edited collections of papers (Kinnear 1994, Angeline and Kinnear 1996), conference proceedings (Koza et al. 1996, 1997), and the World Wide Web (`www.genetic-programming.org`).

Before applying genetic programming to a problem, the user must perform five major preparatory steps. These five steps involve determining

(1) the set of terminals,

(2) the set of primitive functions,

(3) the fitness measure,

(4) the parameters for controlling the run, and

(5) the method for designating a result and the criterion for terminating a run.

The first major step in preparing to use genetic programming is to identify the set of terminals. The terminals can be viewed as the inputs to the as-yet-undiscovered computer program. The set of terminals (along with the set of functions) are the ingredients from which genetic programming attempts to construct a computer program to solve, or approximately solve, the problem.

The second major step in preparing to use genetic programming is to identify the set of functions that are to be used to generate the mathematical expression that attempts to fit the given finite sample of data. Each computer program (i.e., parse tree, mathematical expression, LISP S-expression) is a composition of functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$. Each of the functions in the function set should be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. That is, the function set and terminal set selected should have the closure property so that any possible composition of functions and terminals produces a valid executable computer program. For example, a run of genetic programming will typically employ a protected version of division (returning an arbitrary value such as zero when division by zero is attempted).

The evolutionary process is driven by the *fitness measure*. Each individual computer program in the population is executed and then evaluated, using the fitness measure, to determine how well it performs in the particular problem environment. The nature of the fitness measure varies with the problem. For many problems, fitness is naturally measured by the discrepancy between the result produced by an individual candidate program and the desired result. The closer this error is to zero, the better the program. In a problem of optimal control, the fitness of a computer program may be the amount of time (or fuel, or money, etc.) it takes to bring the system to a desired target state. The smaller the amount, the better. If one is trying to recognize patterns or classify objects into categories, the fitness of a particular program may be measured by accuracy or correlation. For electronic circuit design problems, the fitness measure may involve how closely the circuit's performance (say, in the frequency or time domain) satisfies user-specified design requirements. If one is trying to evolve a good randomizer, the fitness might be measured by means of entropy, satisfaction of the gap test, satisfaction of the run test, or some combination of these factors. For some problems, it may be appropriate to use a multiobjective fitness measure incorporating a combination of factors such as correctness, parsimony (smallness of the evolved program), efficiency (of execution), power consumption (for an electrical circuit), or manufacturing cost (for an electrical circuit).

The primary parameters for controlling a run of genetic programming are the population size, M, and the maximum number of generations to be run, G.

Each run of genetic programming requires specification of a *termination criterion* for deciding when to terminate a run and a method of *result designation*. One frequently used method of result designation for a run is to designate the best individual obtained in any generation of the population during the run (i.e., the *best-so-far individual*) as the result of the run.

In genetic programming, populations of thousands or millions of computer programs are genetically bred for dozens, hundreds, or thousands of generations. This breeding is done using the Darwinian principle of survival and reproduction of the fittest along with a genetic crossover operation appropriate for mating computer programs. A computer program that solves (or approximately solves) a given problem often emerges from this combination of Darwinian natural selection and genetic operations.

Genetic programming starts with an initial population (generation 0) of randomly generated computer programs composed of the given primitive functions and terminals. Typically, the size of each program is limited, for practical reasons, to a certain maximum number of points (i.e. total number of functions and terminals) or a maximum depth (of the program tree). The creation of this initial random population is, in effect, a blind random parallel search of the search space of the problem represented as computer programs.

Typically, each computer program in the population is run over a number of different *fitness cases* so that its fitness is measured as a sum or an average over a variety of representative different situations. These fitness cases sometimes represent a sampling of different values of an independent variable or a sampling of different initial conditions of a system. For example, the fitness of an individual computer program in the population may be measured in terms of the sum of the absolute value of the differences between the output produced by the program and the correct answer to the problem (i.e., the Minkowski distance) or the square root of the sum of the squares (i.e., Euclidean distance). These sums are taken over a sampling of different inputs (fitness cases) to the program. The fitness cases may be chosen at random or may be chosen in some structured way (e.g., at regular intervals or over a regular grid). It is also common for fitness cases to represent initial conditions of a system (as in a control problem). In economic forecasting problems, the fitness cases may be the daily closing price of some financial instrument.

The computer programs in generation 0 of a run of genetic programming will almost always have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat more fit than others. These differences in performance are then exploited.

The Darwinian principle of reproduction and survival of the fittest and the genetic operation of crossover are used to create a new offspring population of individual computer programs from the current population of programs.

The reproduction operation involves selecting a computer program from the current population of programs based on fitness (i.e., the better the fitness, the more likely the individual is to be selected) and allowing it to survive by copying it into the new population.

The crossover operation creates new offspring computer programs from two parental programs selected based on fitness. The parental programs in genetic programming are typically of different sizes and shapes. The offspring programs are composed of subexpressions (subtrees, subprograms, subroutines, building blocks) from their parents. These offspring programs are typically of different sizes and shapes than their parents.

For example, consider the following computer program (presented here as a LISP S-expression):

```
(+ (* 0.234 Z) (- X 0.789)),
```

which we would ordinarily write as

$0.234\,Z + X - 0.789$.

This program takes two inputs (X and Z) and produces a floating point output.

Also, consider a second program:

```
(* (* Z Y) (+ Y (* 0.314 Z))).
```

One crossover point is randomly and independently chosen in each parent. Suppose that the crossover points are the * in the first parent and the + in the second parent. These two crossover fragments correspond to the underlined sub-programs (sub-lists) in the two parental computer programs.

The two offspring resulting from crossover are as follows:

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))

(* (* Z Y) (* 0.234 Z)).
```

Thus, crossover creates new computer programs using parts of existing parental programs. Because entire sub-trees are swapped, the crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. Because programs are selected to participate in the crossover operation with a probability based on fitness, crossover allocates future trials to regions of the search space whose programs contains parts from promising programs.

The mutation operation creates an offspring computer program from one parental programs selected based on fitness. One crossover point is randomly and independently chosen and the subtree occurring at that point is deleted. Then, a new subtree is grown at that point using the same growth procedure as was originally used to create the initial random population.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the old population (i.e., the old generation). Each individual in the new population of programs is then measured for fitness, and the process is repeated over many generations.

The hierarchical character of the computer programs that are produced is an important feature of genetic programming. The results of genetic programming are inherently hierarchical. In many cases the results produced by genetic programming are default hierarchies, prioritized hierarchies of tasks, or hierarchies in which one behavior subsumes or suppresses another.

The dynamic variability of the computer programs that are developed along the way to a solution is also an important feature of genetic programming. It is often difficult and unnatural to try to specify or restrict the size and shape of the eventual solution in advance. Moreover, advance specification or restriction of the size and shape of the solution to a problem narrows the window by which the system views the world and might well preclude finding the solution to the problem at all.

Another important feature of genetic programming is the absence or relatively minor role of preprocessing of inputs and postprocessing of outputs. The inputs, intermediate results, and outputs are typically expressed directly in terms of the natural terminology of the problem domain. The programs produced by genetic programming consist of functions that are natural for the problem domain. The postprocessing of the output of a program, if any, is done by a *wrapper* (*output interface*).

Finally, another important feature of genetic programming is that the structures undergoing adaptation in genetic programming are active. They are not passive encodings (i.e., chromosomes) of the solution to the problem. Instead, given a computer on which to run, the structures in genetic programming are active structures that are capable of being executed in their current form.

Automated programming requires some hierarchical mechanism to exploit, *by reuse* and *parameterization*, the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments. Subroutines do this in ordinary computer programs.

Automatically defined functions can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual programs in the population. Each multi-part program in the population contains one (or more) function-defining branches and one (or more) main result-producing branches. The result-producing branch usually has the ability to call one or more of the automatically defined functions. A function-defining branch may have the ability to refer hierarchically to other already-defined automatically defined functions.

Genetic programming evolves a population of programs, each consisting of an automatically defined function in the function-defining branch and a result-producing branch. The structures of both the function-defining branches and the result-producing branch are determined by the combined effect, over many generations, of the selective pressure exerted by the fitness measure and by the effects of the operations of Darwinian fitness-based reproduction and crossover. The function defined by the function-defining branch is available for use by the result-producing branch. Whether or not the defined function will be actually called is not predetermined, but instead, determined by the evolutionary process.

Since each individual program in the population of this example consists of function-defining branch(es) and result-producing branch(es), the initial random generation must be created so that every individual program in the population has this particular constrained syntactic structure. Since a constrained syntactic structure is involved, crossover must be performed so as to preserve this syntactic structure in all offspring.

Genetic programming with automatically defined functions has been shown to be capable of solving numerous problems (Koza 1994a). More importantly, the evidence so far indicates that, for many problems, genetic programming requires less computational effort (i.e., fewer fitness evaluations to yield a solution with, say, a 99% probability) with automatically defined functions than without them (provided the difficulty of the problem is above a certain relatively low break-even point).

Also, genetic programming usually yields solutions with smaller average overall size with automatically defined functions than without them (provided, again, that the problem is not too simple). That is, both learning efficiency and parsimony appear to be properties of genetic programming with automatically defined functions.

Moreover, there is evidence that genetic programming with automatically defined functions is scalable. For several problems for which a progression of scaled-up versions was studied, the computational effort increases as a function of problem size at a *slower rate* with automatically defined functions than without them. Also, the average size of solutions similarly increases as a function of problem size at a *slower rate* with automatically defined functions than without them. This observed scalability results from the profitable reuse of hierarchically-callable, parameterized subprograms within the overall program.

When single-part programs are involved, genetic programming automatically determines the size and shape of the solution (i.e., the size and shape of the program tree) as well as the sequence of work-performing primitive functions that can solve the problem. However, when multi-part programs and automatically defined functions are being used, the question arises as to how to determine the architecture of the programs that are being evolved. The *architecture* of a multi-part program consists of the number of function-defining branches (automatically defined functions) and the number of arguments (if any) possessed by each function-defining branch. The architecture may be specified by the user, may be evolved using evolutionary selection of the architecture (Koza 1994a), or may be evolved using architecture-altering operations (Koza 1995).

4. Design by Genetic Programming

Genetic programming can be applied to circuits if a mapping is established between the program trees (rooted, point-labeled trees (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 [1990] on using genetic algorithms to evolve neural networks, and the innovative work of Gruau [1992] on using genetic programming to evolve neural networks provide 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. Until these wires are modified, the circuit does not produce interesting output. An electrical circuit is developed by progressively applying the functions in a circuit-constructing program tree to the modifiable wires of the embryo (and, during the developmental process, to new components and modifiable wires). See also Brave 1996.

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 Koza 1994a).

4.1. 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 alternating circuit source VSOURCE. There is a fixed load resistor RLOAD and a fixed source resistor RSOURCE in the embryo. In addition to the fixed components, there is a modifiable wire Z0 between nodes 2 and 3. All development originates from this modifiable wire.

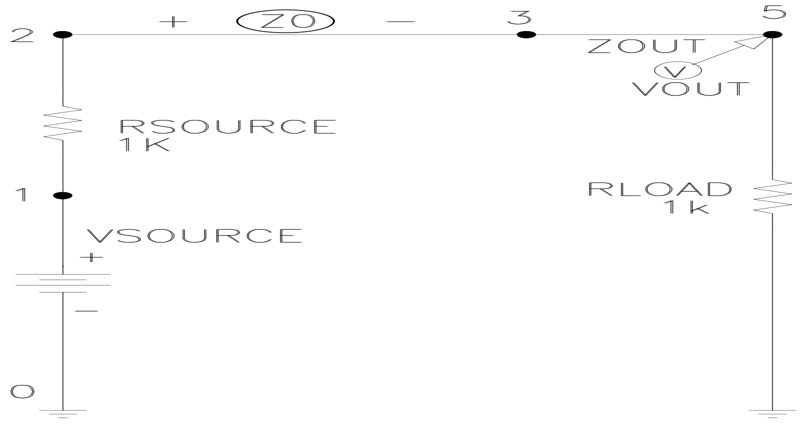

Figure 1 One-input, one-output embryo. **4.2. Component-Creating Functions**

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).

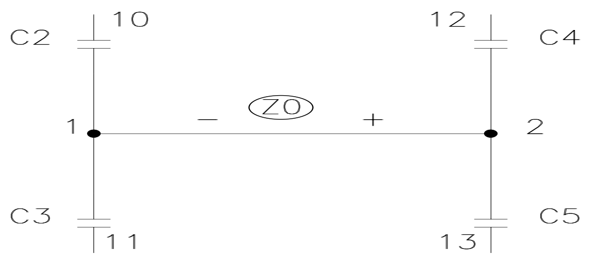

Figure 2 Modifiable wire `Z0`.

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.

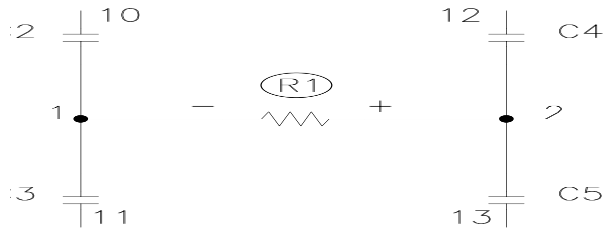

Figure 3 Result of applying the `R` function.

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.

Figure 4 shows the result of applying the Q_3_NPN0 function, thereby creating transistor Q6 in lieu of the resistor R1 of figure 3.

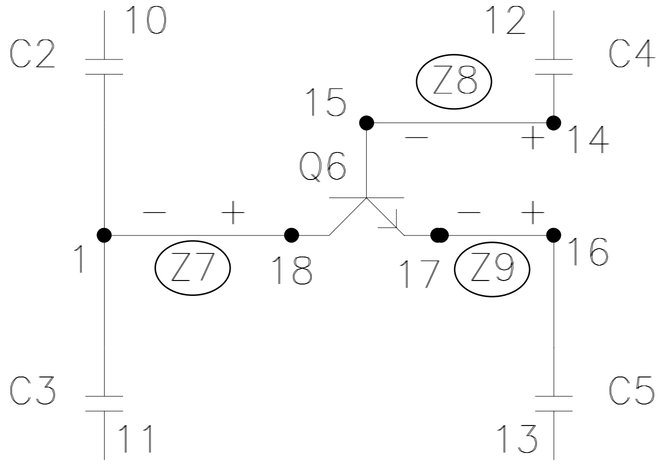

Figure 4 Result of applying transistor-creating Q_3_NPN0 function to resistor R1 of figure 3, thereby transforming it into transistor Q6.

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

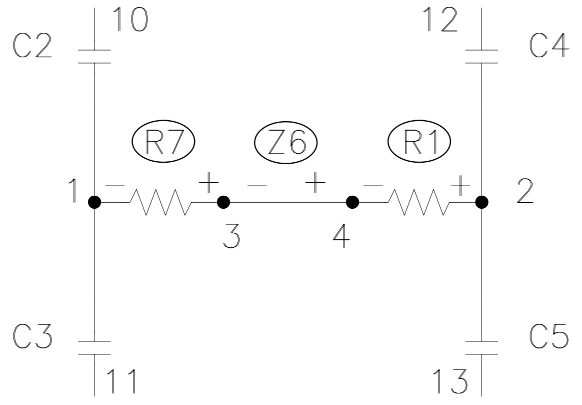

Figure 5 Result after applying the SERIES function to resistor R1 of figure 3, thereby transforming it into resistors R7 and R1 and wire Z6.

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 6 shows the result of applying `PARALLEL0` to the resistor `R1` from figure 3.

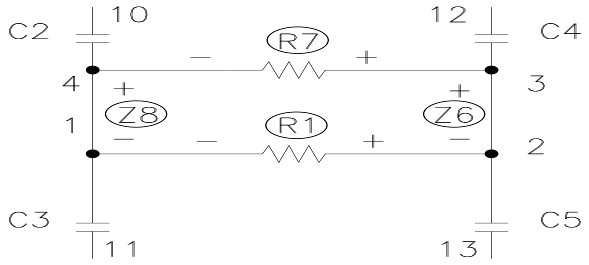

Figure 6 Result of the `PARALLEL0` function. The one-argument polarity-reversing `FLIP` function reverses the polarity of the highlighted component.

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. Figure 7 shows the `T_GND_0` function connecting resistor `R1` of figure 3 to ground.

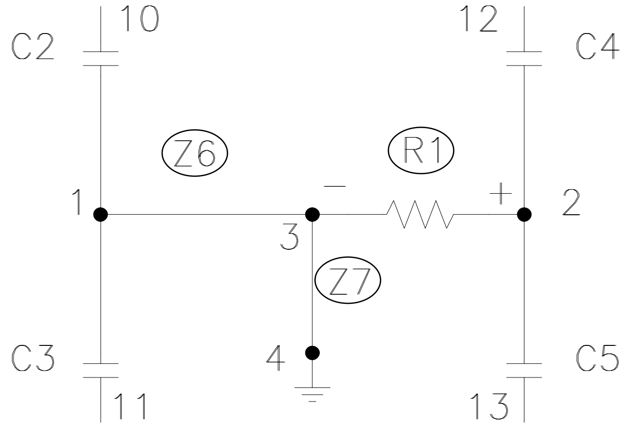

Figure 7 Result of applying the `T_GND_0` function to resistor `R1` of figure 3, thereby creating a connection to ground.

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).

5. 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.

5.1. 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 filter. 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 Koza, Bennett, Andre, and Keane 1997).

5.2. 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 filter problem; 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.

5.3. 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. 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 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

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

5.4. 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 [Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994]. 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.

5.4.1 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 1996b.

5.4.2 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 Koza, Bennett, Lohn, Dunlap, Andre, and Keane 1997b.

5.4.3 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 Koza, Bennett, Lohn, Dunlap, Andre, and Keane 1997a.

5.4.4 Robot Controller Circuit

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 Koza, Bennett, Keane, and Andre 1997.

5.4.5 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 Bennett, Koza, Andre, and Keane 1996.

5.5. Control Parameters

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

5.6. Implementation on Parallel Computer

Each problem was run on a medium-grained parallel Parsytec computer system [Andre and Koza 1996] 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.

6. Results

In all five 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.

6.1. 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 8) 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. This circuit had the recognizable ladder topology [46] of a Butterworth or Chebychev filter (i.e., a composition of series inductors horizontally with capacitors as vertical shunts).

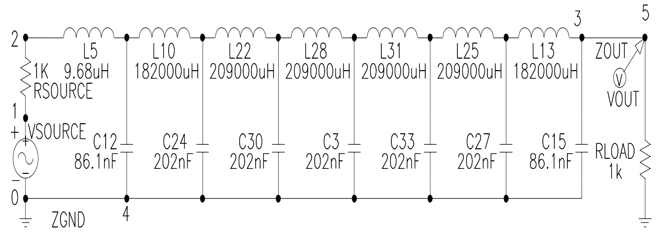

Figure 8 Evolved 7-rung ladder lowpass filter.

Figure 9 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.

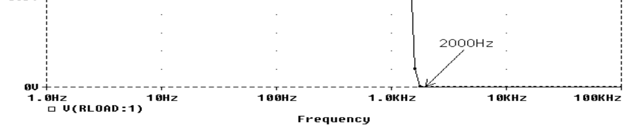

Figure 9 Frequency domain behavior of genetically evolved 7-rung ladder lowpass filter.

6.2. Tri-state Frequency Discriminator

The evolved three-way tri-state frequency discriminator circuit from generation 106 scores 101 hits (out of 101). Figure 10 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.

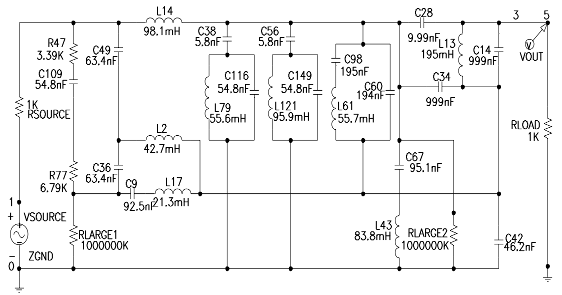

Figure 10 Evolved frequency discriminator.

6.3. Computational Circuit

The genetically evolved computational circuit for the square root from generation 60 (figure 11), 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.

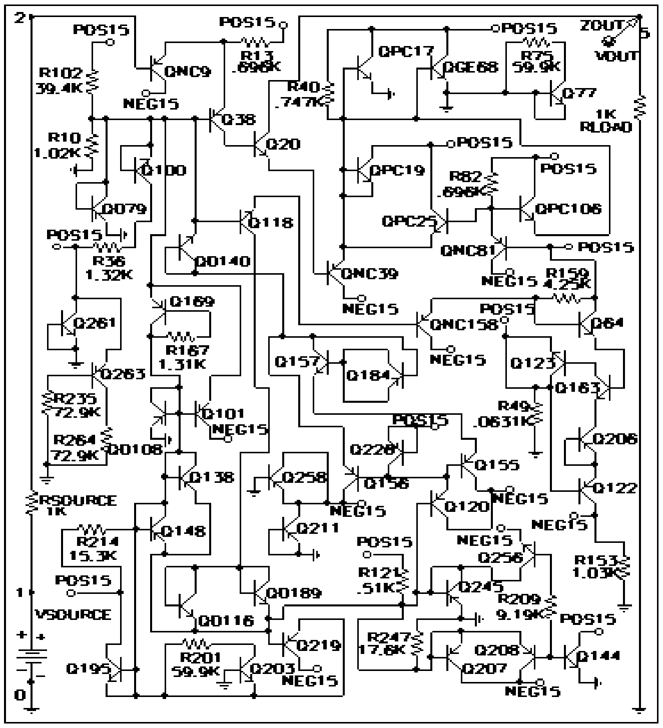

Figure 11 Evolved square root circuit.

6.4. Robot Controller Circuit

The best-of-run time-optimal robot controller circuit (figure 12) 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. The program has one automatically defined function that is called twice (incorporated into the figure).

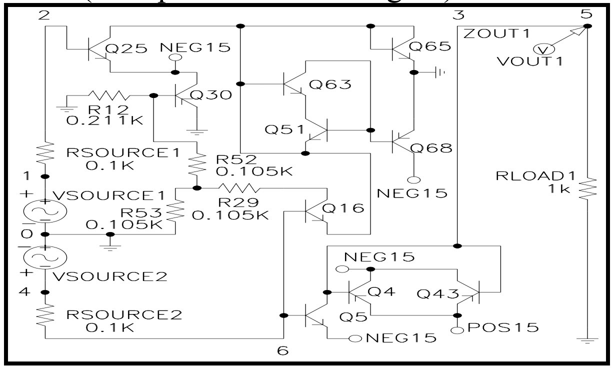

Figure 12 Evolved robot controller. 6.5. 60 dB Amplifier

The best circuit from generation 109 (figure 13) 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.

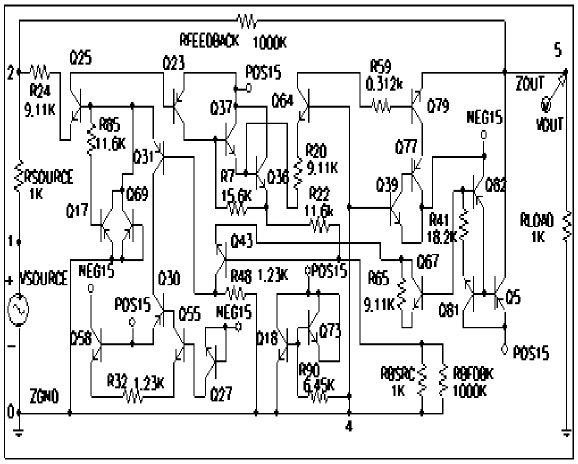

Figure 13 Genetically evolved amplifier.

7. Other Circuits

Numerous other circuits have been similarly designed, including asymmetric bandpass filters [Koza, Bennett, Andre, and Keane 1996c], crossover filters [Koza, Bennett, Andre, and Keane 1996a], double passband filters [Koza, Andre, Bennett, and Keane 1996], amplifiers [Koza, Bennett, Andre, and Keane 1997], a temperature-sensing circuit, and a voltage reference circuit [Koza, Bennett, Andre, Keane, and Dunlap 1997].

8. Conclusion

Genetic programming evolved the topology and sizing of five different prototypical analog electrical circuits. , including a lowpass 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. The problem-specific information required for each of the eight problems is minimal and consists primarily of the number of inputs and outputs of the desired circuit, the types of available components, and a fitness measure that restates the high-level statement of the circuit's desired behavior as a measurable mathematical quantity. All five of these genetically evolved circuits constitute instances of an evolutionary computation technique solving a problem that is usually thought to require human intelligence.

References

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

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

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

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

Bauer, R. J., Jr. 1994. *Genetic Algorithms and Investment Strategies*. John Wiley.

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.

Bhanu, Bir and Lee, Sungkee. 1994. *Genetic Learning for Adaptive Image Segmentation*. Boston: Kluwer Academic Publishers.

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.

Davidor, Yuval. *Genetic Algorithms and Robotics*. Singapore: World Scientific 1991.

Gen, Mitsuo and Cheng, Runwei. 1997. *Genetic Algorithms and Engineering Design*. New York: John Wiley and Sons.

Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

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.

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.

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

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

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

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. 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.

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.

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. Pages 1–10.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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

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.

Man, K. F., Tang, K. S., Kwong, S., and Halang, W. A. 1997. *Genetic Algorithms for Control and Signal Processing*. London: Springer-Verlag.

Michalewicz, Z. 1992. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer-Verlag.

Mitchell, Melanie. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

Ohno, Susumu. *Evolution by Gene Duplication*. New York: Springer-Verlag 1970.

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.

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

Stender, Joachim (editor). 1993. *Parallel Genetic Algorithms*. Amsterdam: IOS Publishing.

Stender, Joachim, Hillebrand, and Kingdon, J. (editors). 1994. *Genetic Algorithms in Optimization, Simulation, and Modeling*. Amsterdam: IOS Publishing.

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.