Toward Evolution of Electronic Animals Using Genetic Programming

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

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

David Andre
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
andre@flamingo.stanford.edu
http://www-leland.stanford.edu/~phred/

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

Abstract

This paper describes an automated process for designing an optimal food-foraging controller for a lizard. The controller consists of an analog electrical circuit that is evolved using the principles of natural selection, sexual recombination, and developmental biology. Genetic programming creates both the topology of the controller circuit and the numerical values for each electrical component.

1. Introduction

Connectionist learning algorithms, reinforcement learning algorithms, genetic algorithms, and other learning algorithms all require, in one way or another, that the system be exposed, in its learning phase, to a non-trivial number of training cases that are representative of the environment.

Researchers in the field of artificial life usually adopt one of two approaches for exposing their system to these training cases. One approach is to simulate the system inside a computer; the other approach is to operate the system in a real-world environment.

An example of the first approach is the familiar simulated robot with errorless sensors that flawlessly executes operations in a sanitized environment in discrete time and space. Although such simulations can be conducted at high speeds within a computer, they may have little resemblance to the real-world environment.

An example of the second approach is an actual physical robot with noisy sensors that imperfectly executes operations in a realistic environment. However, the time required for actual operation in the real world precludes exposing the system to any significant number of training episodes. For example, in a novel experiment, Floreano and Mondada (1994) ran the genetic algorithm on a fast workstation to evolve a control strategy for an obstacle-avoiding robot. The fitness of an individual strategy in the population within a particular generation of the run was determined by executing a physical robot tethered to the workstation for 30 seconds in real time. This experiment was necessarily severely limited because there are only 2,880 30-second intervals in a day. Consequently, a run involving a population of only 80 individuals and only 100 generations required about three days. One can contemplate shortening the time for each episode by perhaps one order of magnitude and one can also contemplate simultaneously operating more than one physical robot in parallel (at increasing financial investment). However, it is difficult to see how this approach can offer any realistic possibility of being scaled up by the *many* orders of magnitude necessary to undertake significant learning or evolution. On the other hand, there will likely be increases of many orders of magnitude in computer speed because of both speedups in microprocessors and speedups from parallelization.

This paper proposes that a way to get the best of both of the above two approaches is to do the simulation inside a computer using a highly realistic simulation employing the very same parts that would be used in a realistic system in the physical world. Specifically, we describe how we used a currently available accurate analog electrical simulator in conjunction with genetic programming to evolve a controller composed of analog electrical parts.

2. Optimal Food-Foraging Strategy

The *Anolis* lizard (figure 1) of the Caribbean is a "sit and wait" predator that perches head-down on tree trunks and scans the ground for edible insects.

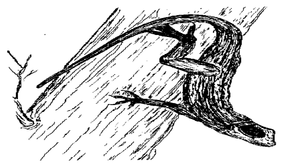

Figure 1 *Anolis* lizard perched on a tree trunk.

Roughgarden (1995) shows that the food-foraging strategy that yields the most food calls for the lizard to chase an insect alighting at distance, x, within its viewing area if

$$x < \sqrt{(\frac{3v}{\pi a})},$$

where abundance, a, is the number of insects per square meter per second and where v is the lizard's sprint velocity. (See also Koza, Rice, and Roughgarden 1992).

3. Electrical Implementation

A foraging strategy can be realized by an electrical circuit (figure 2) whose input comes from the input neuron of the lizard's visual system and whose output goes to an output neuron that causes the lizard to chase an insect. The visual neuron may generate a signal whose frequency is proportional to the logarithm of the insect's distance, *x* (with, say, 1000 Hertz corresponding to 8 meters). The output neuron may receive a voltage (say, 1 volt) that activates the lizard. The food-foraging problem can be viewed as a problem of designing a circuit (figure 2).

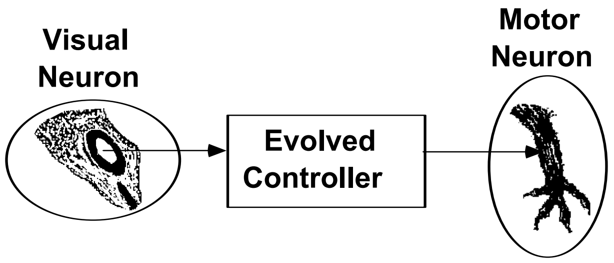

Figure 2 Controller has input at lizard's visual neuron and output at lizard's motor neuron.

Electrical circuits consist of a variety of different types of components, including resistors, capacitors, inductors, diodes, transistors, and energy sources. The problem of circuit synthesis involves designing an electrical circuit that satisfies user-specified design goals. Circuits receive input signals from zero, one, or more input sources and produce output signals at one or more output ports (probe points). In designing a circuit, the goal is to achieve certain desired values of one or more observable (or calculable) quantities involving the output(s) of the circuit (in relation to its inputs). A complete specification of an electrical circuit includes both its topology and the sizing of all its components. The *topology* of a circuit consists of the number of components in the circuit, the type of each component, and a list of the connections between the components. The *sizing* of a circuit consists of the component value(s) (typically numerical) associated with each component.

Electrical engineers will recognize that a lowpass filter can implement the above optimal food-foraging strategy. Specifically, the desired filter might have a passband below 1,000 Hertz and a stopband above 2,000 Hz. The passband voltage might be between, say, 970 millivolts and 1 volt (i.e., a passband ripple of 0.3 decibels or less) and the stopband voltage might be between 0 volts and 1 millivolts (i.e., a stop band attenuation of at least 60 decibels). These design requirements can be satisfied by an elliptic (Cauer) filter of order 5, with a reflection coefficient of 20%, and modular angle of 30 degrees. The circuit is assumed to be driven from a AC input source with 2 volt amplitude with an internal (source) resistance of 1,000 Ohms and a load resistance of 1,000 Ohms.

SPICE (an acronym for Simulation Program with Integrated Circuit Emphasis) is a massive 217,000-line program written over several decades at the University of California at Berkeley for the accurate simulation of analog, digital, and mixed analog/digital electrical circuits (Quarles et al. 1994). SPICE performs various types of analysis on circuits containing various circuit elements. The input to a SPICE simulation consists of a netlist describing the circuit and certain commands concerning the type of analysis to be performed and output to be produced.

4. Genetic Programming

Genetic programming is an extension of John Holland's genetic algorithm (1975) in which the population consists of computer programs of varying sizes and shapes (Koza 1992, 1994a, and 1994b; Koza and Rice 1992).

5. Cellular Encoding of Neural Networks

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

6. Analog Circuit Synthesis

Considerable progress has been made in automating the design of certain categories of purely digital circuits. Hemmi, Mizoguchi, and Shimohara (1994) and Higuchi et al. (1993) have employed genetic methods to the design of digital circuits using a hardware description language (HDL).

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

7. The Mapping between Program Trees and Electrical Circuits

Genetic programming breeds a population of rooted, point-labeled trees (i.e., graphs without cycles) with ordered branches. There is a considerable difference between the kind of trees bred by genetic programming and the labeled cyclic graphs encountered in the world of electrical circuits.

Electrical circuits are cyclic graphs in which *every* line belongs to a cycle (i.e., there are no loose wires or dangling components). The lines of a graph that represents a circuit are each labeled. The primary label on each line gives the type of an electrical component. The secondary label(s), if any, on each line give the value(s) of the component(s), if any. One numerical value is sufficient to specify certain components (e.g., resistors); none are required for diodes; and many are required for a sinusoidal voltage source.

Genetic programming can be applied to circuits if a mapping is established between the kind of point-labeled trees found in the world of genetic programming and the line-labeled cyclic graphs employed in the world of circuits. In our case, developmental biology provides the motivation for this mapping. The growth process used herein begins with a very simple embryonic electrical circuit and builds a more complex circuit by progressively executing the functions in a circuit-constructing program tree. The result is the topology of the circuit, the choice of types of components that are situated at each location within the topology, and the sizing of all the components.

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

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

Both the random program trees in the initial population (generation 0) and all random subtrees created by the mutation operation in later generations are created so as to conform to this constrained syntactic structure. This constrained syntactic structure is preserved by using structure-preserving crossover with point typing (Koza 1994a).

8. The Embryonic Electrical Circuit

The embryonic circuit used on a problem depends on the number of input signals and the number of output signals.

The embryonic circuit used herein contains one input signal, one output (probe point), a fixed source resistor, and a fixed load resistor, and two modifiable wires. The two modifiable wires (Z0 and Z1) each initially possess a writing head (i.e., are highlighted with a circle in figure 3). A circuit is progressively developed by modifying the component to which a writing head is pointing in accordance with the functions in the circuit-constructing program tree. Each connection-modifying and component-creating function in the program tree modifies the developing circuit in a particular way and each also specifies the future disposition of the writing head(s).

Figure 3 shows the embryonic circuit used for the one-input, one-output filter circuit discussed herein. The energy source is a 2 volt voltage source VSOURCE whose negative (−) end is connected to node 0 (ground) and whose positive (+) end is connected to node 1. There is a fixed 1000-Ohm source resistor RSOURCE between nodes 1 and 2. There is a modifiable wire Z1 between nodes 2 and 3 and another modifiable wire Z0 between nodes 3 and 4. There are circles around Z0 and Z1 to indicate that the two writing heads point to these modifiable wires. There is a fixed isolating wire ZOUT between nodes 3 and 5, a voltage probe labeled VOUT at node 5, and a fixed 1000-Ohm load resistor RLOAD between nodes 5 and ground. There is an isolating wire ZGND between nodes 4 and 0 (ground). All of the above elements of this embryonic circuit (except Z0 and Z1) are fixed and not subject to modification during the process of developing the circuit. All subsequent development of the circuit originates from writing heads. Note that the output of the embryonic circuit is a constant zero volt signal VOUT at node 5.

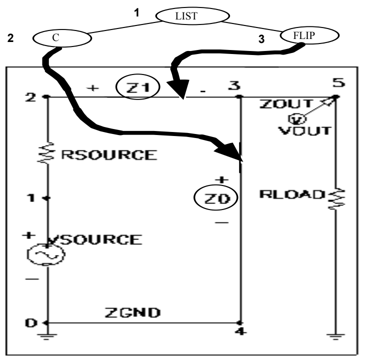

Figure 3 One-input, one-output embryonic electrical circuit.

The domain knowledge that went into this embryonic circuit consisted of the facts that (1) the embryo is a circuit, (2) the embryo has one input and one output, and (3) there are modifiable connections between the output and the source and between the output and ground.

A circuit is developed by modifying the component to which a writing head is pointing in accordance with the associated function in the circuit-constructing program tree. The figure shows a capacitor-creating C function (described later) and a polarity-reversing FLIP function (described later) just below the connective LIST function at the root of the program tree. The figure also shows a writing head pointing from the C function to modifiable wire Z0 and pointing from the FLIP function to modifiable wire Z1. This C function will cause Z0 to be changed into a capacitor and the FLIP function will cause the polarity of modifiable wire Z1 to be reversed.

9. Component-Creating Functions

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

Each component-creating function inserts a component into the developing circuit and assigns component value(s) to the inserted component. Each component-creating function spawns one or more writing heads (through its construction-continuing subtrees). The construction-continuing subtree of each component-creating function points to a successor function or terminal in the circuit-constructing program tree.

The arithmetic-performing subtree of a component-creating function consists of a composition of arithmetic functions (addition and subtraction) and random constants (in the range –1.000 to +1.000). The arithmetic-performing subtree specifies the numerical value of the component by returning a floating-point value that is, in turn, interpreted as the value for the component in a range of 10 orders of magnitude (using a unit of measure that is appropriate for the particular type of component involved). The floating-point value is interpreted as the value of the component in the following way: If the return value is between –5.0 and +5.0, U is equated to the value returned by the subtree. If the return value is less than –100 or greater than +100, U is set to zero. If the return value is between –100 and –5.0, U is found from the straight line connecting the points (–100, 0) and (–5, -5). If the return value is between +5.0 and +100, U is found from the straight line connecting (5, 5) and (100, 0). The value of the component is 10^U in a unit that is appropriate for the type of component. This mapping gives the component a value within a range of 10 orders of magnitude centered on a certain value.

9.1. The C Function

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

9.2. The L Function

The two-argument inductor-creating L function causes the highlighted component to be changed into an inductor. The value of the inductor is in micro-Henrys within a range of plus or minus 5 orders of magnitude centered on 1 μH.

9.3. Other Component-Creating Functions

Numerous other component-creating functions can be employed in this process. We describe one other function for illustrative purposes (even though it is not used in solving the optimal food-foraging problem for the lizard).

Figure 4 shows a resistor R1 (with a writing head) connecting nodes 1 and 2 of a partial circuit.

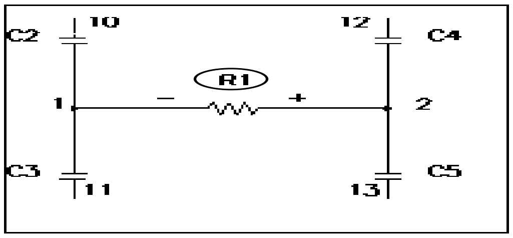

Figure 4 Circuit with resistor R1.

The functions in the group of three-argument transistor-creating QT functions cause a transistor to be inserted in place of one of the nodes to which the highlighted component is currently connected (while also deleting the highlighted component). Each QT function also creates five new nodes and three new modifiable wires. After execution of a QT function, there are three writing heads that point to three new modifiable wires.

Figure 5 shows the result of applying the QT0 function to resistor R1 of figure 4, thereby creating a transistor Q6.

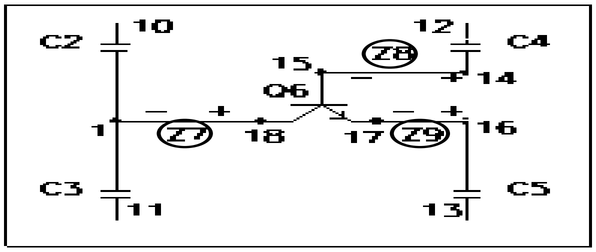

Figure 5 Result of applying QT0 function.

10. Connection-Modifying Functions

The topology of the circuit is determined by the connection-modifying functions.

10.1. The FLIP Function

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

10.2. SERIES Division Function

The three-argument SERIES division function operates on one highlighted component and creates a series composition consisting of the highlighted component, a copy of the highlighted component, one new modifiable wire, and two new nodes. After execution of the SERIES function, there are three writing heads pointing to the original component, the new modifiable wire, and the copy of the original component. Figure 6 shows the result of applying the SERIES function to resistor R1 of figure 4.

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

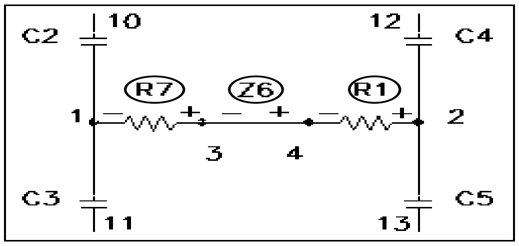

Figure 6 Result of applying SERIES.

10.3. Parallel Division PSS Function

The four-argument parallel division function PSS operates on one highlighted component to create a parallel composition consisting of the original highlighted component, a duplicate of the highlighted component, two new wires, and two new nodes. After execution of PSS, there are four writing heads. They point to the original component, the two new modifiable wires, and the copy of the original component.

First, the parallel division function PSS creates two new nodes, 3 and 4. Second, PSS inserts a duplicate of the highlighted component (including all of its component values) between the new nodes 3 and 4 (with the negative end of the duplicate connected to node 4 and the positive end of the duplicate connected to 3). Third, PSS creates a first new wire Z6 between the positive (+) end of R1 (which is at original node 2) and first new node, 3. Fourth, PSS creates a second new wire Z8 between the negative (-) end of R1 (which is at original node 1) to second new node, 4.

Figure 7 shows the results of applying the PSS function to resistor R1 from figure 4. The negative end of the new component is connected to the smaller numbered component of the two components that were originally connected to the negative end of the highlighted component. Since C4 bears a smaller number than C5, new node 3 and new wire Z6 are located between original node 2 and C4. Since C2 bears a smaller number than C3, new node 4 and new wire Z8 are located between original node 1 and C2.

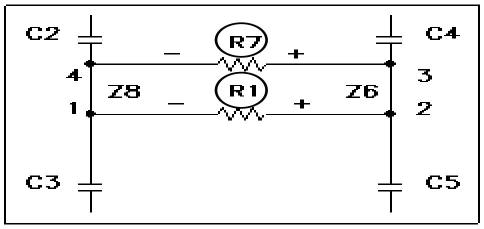

Figure 7 Result of applying PSS.

10.4. VIA and GND Functions

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

The VIA functions create a series composition consisting of two wires that each possess a successor writing head and a numbered port (called a *via*) that possesses no writing head. The port is connected to a designated one of eight imaginary layers (numbered from 0 to 7) of an imaginary silicon wafer. If one or more parts of the circuit connect to a particular layer, all such parts become electrically connected as if wires were running between them.

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

10.5. The NOP Function

The one-argument NOP function has no effect on the highlighted component; however, it delays activity on the developmental path on which it appears in relation to other developmental paths. After execution of NOP, one writing head points to the original highlighted component.

10.6. The END Function

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

10.7. Other Connection-Modifying Functions

Numerous other connection-modifying functions can be employed in this process. We describe two other functions for illustrative purposes (not used in the problem at hand).

The functions in the group of three-argument Y division functions operate on one highlighted component (and one adjacent node) and create a Y-shaped composition consisting of the highlighted component, two copies of the highlighted component, and two new nodes. The Y functions insert the two copies at the "active" node of the highlighted component. For the Y1 function, the active node is the node to which the negative end of the highlighted component is connected. Figure 8 shows the result of applying Y1 to resistor R1 of figure 4.

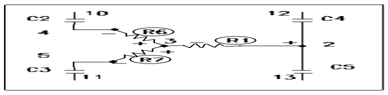

Figure 8 Result of applying the Y1 function.

The functions in the group of six-argument DELTA functions operate on one highlighted component by eliminating it (and one adjacent node) and creating a triangular Δ−shaped composition consisting of three copies of the original highlighted component (and all of its component values), three new modifiable wires, and five new nodes. Figure 9 illustrates the result of applying the DELTA1 division function to resistor R1 of figure 4 when the active node (node 1) is of degree 3.

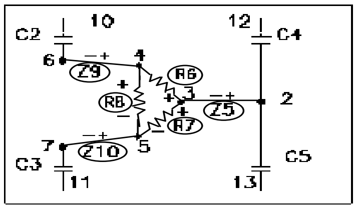

Figure 9 Result of applying DELTA1 function.

11. Preparatory Steps

Since the problem of designing the lowpass LC filter calls for a one-input, one-output circuit with a source resistor and a load resistor, the embryonic circuit of figure 3 is suitable for this problem.

Since the embryonic circuit starts with two writing heads, each program tree has two result-producing branches joined by a LIST function. There are no automatically defined functions. The terminal set and function set for both result-producing branches are the same. Each result-producing branch is created in accordance with the constrained syntactic structure that uses the left (first) argument(s) of each component-creating function to specify the numerical value of the component. The numerical value is created by a composition of arithmetic functions and random constants in this arithmetic-performing subtree. The right (second) argument of each component-creating function is then used to continue the program tree.

In particular, the function set, $\mathcal{F}_{\text{aps}}$, for an arithmetic-performing subtree is

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

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

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

where $\leftarrow$ represents floating-point random constants between –1.000 and +1.000.

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

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

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

$\mathcal{T}_{\text{ccs}} = \{$`END`$\}$.

Note that all of the above is applicable to any LC circuit involving one input and one output.

The user-supplied fitness measure drives the evolutionary process. In general, the fitness measure may incorporate any calculable characteristic or combination of characteristics of the circuit, including the circuit's behavior in the time domain, its behavior in the frequency domain, its power consumption, or the number, cost, or surface area occupied by its components.

The evaluation of fitness for each individual circuit-constructing program tree in the population begins with its execution. This execution applies the functions in the program tree to the very simple embryonic circuit, thereby developing it into a fully developed circuit. A netlist describing the circuit is then created. The netlist identifies each component of the circuit, the nodes to which that component is connected, and the value of that component. Each circuit is then simulated to determine its behavior using the 217,000-line SPICE simulator (modified to run as a submodule within our genetic programming system).

Since we are designing a filter, the focus is on the behavior of the circuit in the frequency domain. SPICE is requested to perform an AC small signal analysis and to report the circuit's behavior for each of 101 frequency values chosen from the range between 10 Hz to 100,000 Hz (in equal increments on a logarithmic scale). Fitness is measured in terms of the sum, over these 101 fitness cases, of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit at the probe point `VOUT` at isolated node `5` and the target value for voltage. The smaller the value of fitness, the better (with zero being best). 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 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 does not penalize ideal values; it slightly penalizes every acceptable deviation; and it heavily penalizes every unacceptable deviation.

The procedure for each of the 61 points in the 3-decade interval from 1 Hz to 1,000 Hz is as follows: If the voltage is between 970 millivolts and 1,000 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 1.0. If the voltage is less than 970 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 10.0. This arrangement reflects the fact that the ideal voltage in the passband is 1.0 volt, the fact that a 30 millivolt shortfall is acceptable, and the fact that a voltage below 970 millivolts is not acceptable.

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

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

The deviation is overlooked for each of the 5 points in the interval between 1,000 Hz and 2,000 Hz (i.e., the "don't care" band). Hits are defined as the number of fitness cases for which the voltage is acceptable or ideal or which lie in the "don't care" band. Thus, the number of hits ranges from a low of 5 to a high of 101 for this problem.

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

The population size, M, is 320,000. The percentage of genetic operations on each generation was 89% crossovers, 10% reproductions, and 1% mutations. A maximum size of 200 points was established for each of the two result-producing branches in each overall program. The other parameters for controlling the runs of genetic programming were the default values of Koza 1994a (appendix D).

This problem was run on a medium-grained parallel Parystec computer system consisting of 64 Power PC 601 80 MHz processors arranged in a toroidal mesh with a host PC Pentium type computer (running Windows). The so-called *distributed genetic algorithm* was used with a subpopulation (deme) size of Q = 5,000 at each of D = 64 demes. On each generation, four boatloads of emigrants, each consisting of B = 2% (the migration rate) of the node's subpopulation (selected on the basis of fitness) were dispatched to each of the four toroidally adjacent processing nodes. Details of the parallel implementation of genetic programming can be found in Andre and Koza 1996.

12. Results

12.1. A Circuit with the "Ladder" Topology

The best individual program tree from generation 0 of run A has a fitness of 58.71, scores 51 hits.

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

SPICE cannot simulate about two-thirds of the programs of generation 0 for this problem. However, the percentage of unsimulatable circuits drops rapidly as new offspring are created using Darwinian selection, crossover, and mutation. The percentage of unsimulatable programs drops to 33% by generation 10, and 0.3% by generation 30.

The best individual program tree of generation 32 has a fitness of 0.00781 and scores 101 hits. Figure 10 shows the best circuit from generation 32 from run A. It has a recognizable seven-rung ladder topology of a Butterworth or Chebychev filter. It also possesses repeated values of various inductors (in series horizontally across the top of the figure) and capacitors (vertical shunts). Figure 11 shows the behavior in the frequency domain of this circuit. The circuit delivers a voltage of virtually 1 volt between 1 Hz and 1,000 Hz and virtually suppresses the voltage above 2,000 Hz.

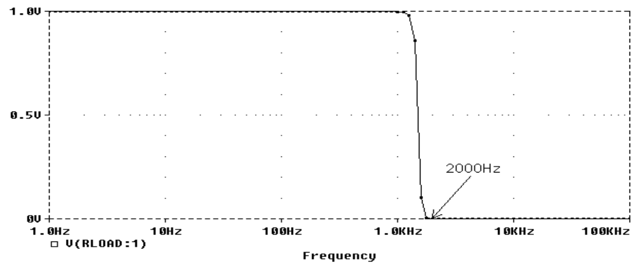

Figure 11 Frequency domain behavior.

12.2. A Circuit with "Bridged T" Topology

Different runs of genetic programming produce different results. Figure 12 shows a fully compliant best circuit from generation 64 of another run (run B). In this circuit, inductor L14 forms a recognizable *bridged T* arrangement in conjunction with C3 and C15 and L11. The bridged T arrangement is a different topology than the ladder topology.

Note that if we disregard C12 (whose 0.338 nF size is insignificant in relation to the 127 nF size of C24 and C21), there are three *π sections* to the left of the "bridged T." Each π section is a π-shaped segment consisting of a 127 nF capacitor as the left leg of the π, an inductor at the top, and another 127 nF capacitor as the right leg of the π).

12.3. A Circuit with a Novel Topology

Figure 13 shows a fully compliant circuit from generation 212 of another run (run C) with a novel topology that no electrical engineer would be likely to create.

13. Related Work and Future Work

We have also genetically designed a difficult-to-design asymmetric bandpass filter (Koza, Andre, Bennett, and Keane 1996), a crossover (woofer and tweeter) filter, an amplifier and other circuits (Koza, Bennett, Andre, and Keane 1996).

14. Conclusions

We have described an automated design process for designing analog electrical circuits based on the principles of natural selection, sexual recombination, and developmental biology. The paper described how genetic programming evolved the design of a low-pass filter that is the solution to the problem of finding the optimal foraging strategy for a lizard.

Acknowledgements

Tom L. Quarles advised us about SPICE. Figure 1 is courtesy of Jonathan Roughgarden from *Theory of Population Genetics and Evolutionary Ecology: An Introduction* (1979). Simon Handley made helpful comments on drafts of this paper.

Bibliography

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

Floreano, Dario and Mondada, Francesco. 1994. Automatic creation of an autonomous agent: Evolution of a Neural-Network Drive Robot. In Cliff, Dave, Husbands, Philip, Meyer, Jean-Arcady, and Wilson, Stewart W. (editors). 1994. *From Animals to Animats 3 Proceedings of the Third International Conference on Simulation of Adaptive Behavior*. Pages 421–430.

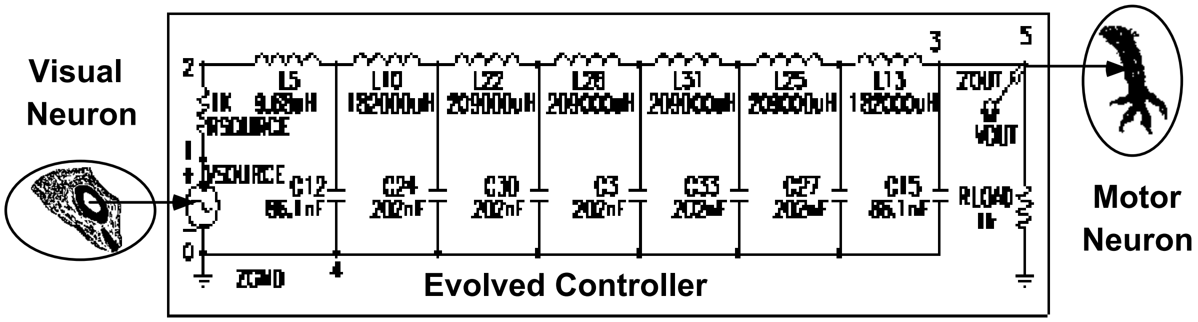

Figure 10 Best-of-run seven-rung "ladder" circuit from generation 32 of run A.

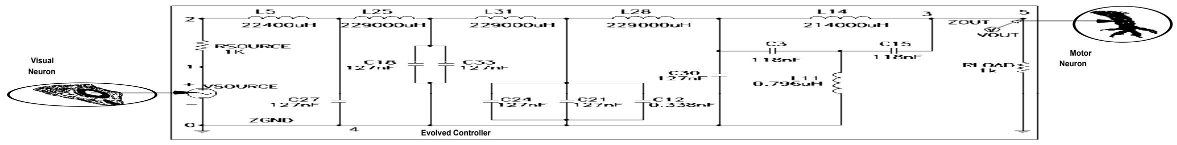

Figure 12 "Bridged T" circuit from generation 64 of run B.

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.

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

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

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

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

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

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

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

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

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

Koza, John R., Rice, James P., and Roughgarden, Jonathan. 1992. Evolution of food-foraging strategies for the Caribbean *Anolis* lizard using genetic programming. *Adaptive Behavior*. 1(2) 47-74.

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

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

Roughgarden, Jonathan. *Anolis Lizards of the Caribbean: Ecology, Evolution, and Plate Tectonics*. Oxford University Press 1995.

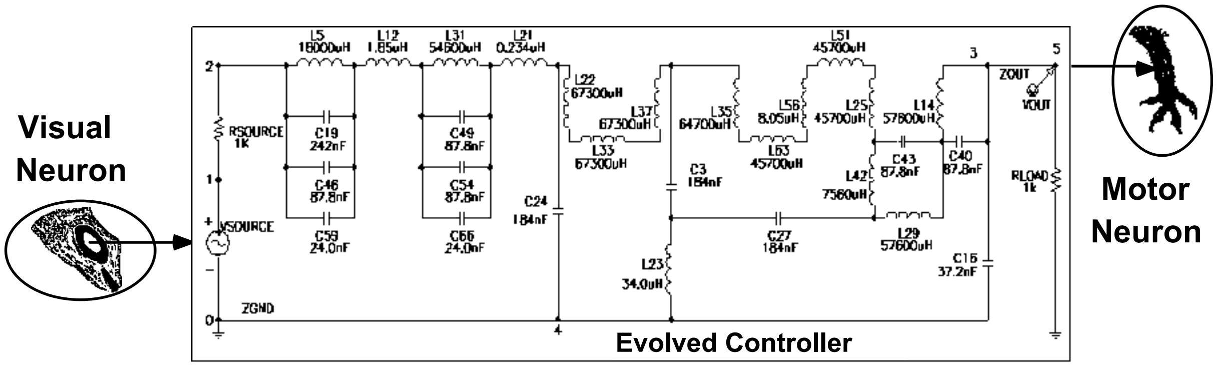

Figure 13 Novel topology of 100% compliant circuit from generation 212 of run C.