Automated WYWIWYG Design of Both the Topology and Component Values of Electrical Circuits 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

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

ABSTRACT

This paper describes an automated process for designing electrical circuits in which "What You Want Is What You Get" ("WYWIWYG" – pronounced "wow-eee-wig"). The design process uses genetic programming to produce both the topology of the desired circuit and the sizing (numerical values) for all the components of a circuit. Genetic programming successfully evolves both the topology and the sizing for an asymmetric bandpass filter that was described as being difficult-to-design in a leading electrical engineering journal. This evolved circuit is another instance in which a genetically evolved solution to a non-trivial problem is competitive with human performance.

1. Introduction

The problem of circuit synthesis involves designing an electrical circuit that satisfies user-specified design goals.

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.

Considerable progress has been made in automating the design of certain categories of purely digital circuits; however, analog circuits and mixed analog-digital circuits are not as amenable to automation (Rutenbar 1993).

Once the user has specified the design goals for an electronic 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 circuit synthesis.

This paper shows how genetic programming was used to design an asymmetric bandpass filter that was described as difficult-to-design in an article in the *Analog Integrated Circuits and Signal Processing* journal (Nielsen 1995).

2. Automated Analog Design Tools

Existing techniques to automate the design process for analog and mixed analog-digital circuits typically require the user to supply a suitable topology; require the user to supply a reasonably good prototype circuit as a starting point; require repeated interactive intervention by the user; limit the user to certain specialized types of circuits; or are otherwise highly constrained. Examples include IDAC (Degrauwe 1987); OASYS (Harjani, Rutenbar, and Carley 1989); OPASYN (Koh, Sequin, and Gray 1990); SEAS (Ning, Kole, Mouthaan, and Wallings 1992); the expert design knowledge system of Maulik, Carley, and Rutenbar (1992); the works of Hemmi, Mizoguchi, and Shimohara (1994); Mizoguchi, Hemmi, and Shimohara 1994; and Higuchi et al. (1993); the "C-T filter design tool" of Nielsen (1995); and DARWIN (Kruiskamp and Leenaerts 1995). In DARWIN, circuit topologies are limited to a preestablished hand-designed set of 24 suitable opamp topologies while component values are adjusted by a genetic algorithm operating on a binary character string.

3. The Mapping between Electrical Circuits and Program Trees

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, 1994b; Koza and Rice 1992).

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 represent a circuit are each labeled. The primary label on each line gives the type of an electrical component (e.g., resistor). The secondary label(s), if any, give the component value(s).

Genetic programming can be applied to circuit synthesis if a mapping is established between the kind of point-labeled trees found in the world of genetic programming and the line-labeled cyclic graphs employed in the world of circuits. Developmental biology provides the motivation for this mapping. In *Cellular Encoding of Genetic Neural Networks*, Frederic Gruau (1992) described a clever technique, called *cellular encoding*, in which genetic programming is used to evolve the architecture of a neural network. Each individual program tree in the population is a specification for developing a complete neural network from a very simple embryonic neural network (consisting of a single neuron). Genetic programming is then applied to populations of network-constructing program trees in order to evolve a neural network capable of solving a problem. The growth process used herein for electrical circuits similarly 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 of this process is the topology of the circuit, the choice of the types of components that are situated at each location within the topology, and the sizing of 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 value (sizing) for each such component, and possibly (3) automatically defined functions.

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

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

4. The Problem

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

In the *Analog Integrated Circuits and Signal Processing* journal, Ivan Riis Nielsen (1995) described stringent and asymmetric specifications for a difficult-to-design bandpass filter. Nielsen's bandpass filter (1995) is targeted for a modem application where one band of frequencies (31.2 to 45.6 kilohertz) must be isolated from another (69.6 to 84.0 KHz). Using a standard bandpass filter on this problem would require a tenth-order elliptic function.

A *decibel* (dB) is a unitless measure of relative voltage that is defined as 20 times the common logarithm of the ratio between the voltage at a particular probe point and a reference voltage (which, herein, is always 1.0 volts).

Nielsen specified that it would be *ideal* if the relative voltage within the passband were in the narrow region between –0.6 dB and 0.6 dB (i.e., the *passband ripple* around 0 dB is less than 0.6 decibels) and all the relative voltages outside the passband were below –120 dB (i.e., the *stopband attenuation* were at least 120 dB). These ideal characteristics are depicted by the dark region in figures 11, 12, and 13 (where the height of the ripple band is exaggerated).

Nielsen defined a set of *acceptable* characteristics (depicted by the light shading in figures 11, 12, and 13). Because of the importance of isolating the band of frequencies between 69.6 and 84.0 KHz, the attenuation there should be at least 73 dB (i.e., the relative voltage is below –73 dB). Less stringency is demanded elsewhere. The attenuation for frequencies below 20 KHz should be at least 38 dB (i.e., the relative voltage is below –38 dB). The relative voltages in the frequency band between 20 KHz and 31.2 KHz and in the band between 45.6 KHz and 69.6 KHz should be below 0 dB. The relative voltages in the band above 84.0 KHz should be below –55 dB. Nielsen's filter has a fixed 1 Ohm source resistor RSOURCE and a fixed 0.1 Ohm load resistor RLOAD.

5. Embryonic Electrical Circuit

The embryonic circuit used on a particular problem depends on the number of input signals and the number of output signals (probe points).

The embryonic circuit used herein contains one input signal, one output (probe point), Nielsen's fixed source resistor, and Nielsen's 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 1). 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).

The bottom part of figure 1 shows the embryonic circuit used for a one-input, one-output circuit. The energy source is a 22 volt sinusoidal voltage source VSOURCE whose negative (−) end is connected to node 0 (ground) and whose positive (+) end is connected to node 1. There is a fixed 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 is a fixed isolating wire ZOUT between nodes 3 and 5, a voltage probe labeled VOUT at node 5, and a fixed 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 forever; they are 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.

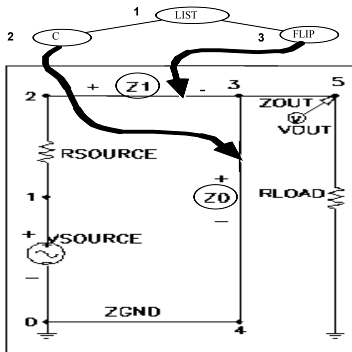

Figure 1 One-input, one-output embryonic electrical circuit.

The domain knowledge that went into this embryonic circuit is minimal. It 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 and a polarity-reversing `FLIP` function just below the connective `LIST` function at the root of the program tree. The figure also shows one writing head pointing from the `C` function to modifiable wire Z0 and another writing head pointing from the `FLIP` function to modifiable wire Z1. As will be described below, 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.

6. Component-Creating Functions

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

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

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

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.

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.

We describe one other function (not used in the main problem herein) to illustrate that numerous other component-creating functions may be used in this process.

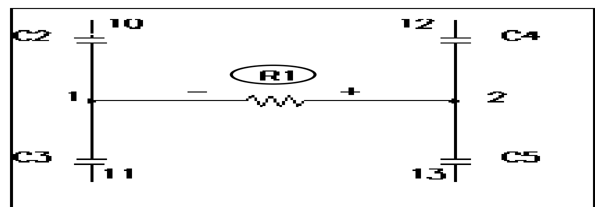

Figure 2 Circuit with resistor R1.

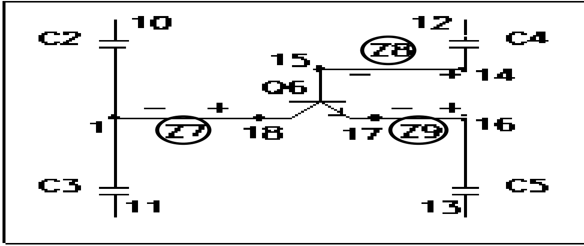

Figure 3 Result of applying `QT0` function.

Figure 2 shows a resistor R1 (with a writing head) connecting nodes 1 and 2 of a partial circuit. Each of the 12 context-sensitive functions in the group of three-argument QT ("transistor") functions (called QT0, ..., QT11) causes 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 3 shows the result of applying the QT0 function to resistor R1 of figure 2, thereby creating transistor Q6.

7. Connection-Modifying Functions

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

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

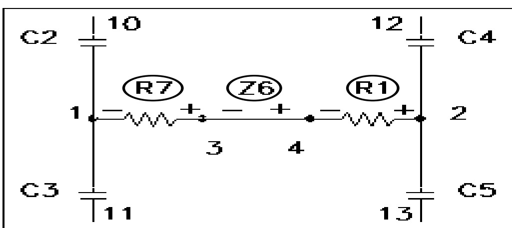

Figure 4 Result of applying `SERIES`.

The three-argument SERIES division function operates on one highlighted component and creates a series composition consisting of the highlighted component, a copy of the highlighted component, one new modifiable wire, and two new nodes. After execution of the SERIES function, there are three writing heads pointing to the original component, the new modifiable wire, and the copy of the original component. Figure 4 shows the result of applying the SERIES division function to resistor R1 from figure 2. First, the SERIES function creates two new nodes, 3 and 4. Second, SERIES disconnects the negative end of the original component (R1) from node 1 and connects this negative end to the first new node, 4 (while leaving its positive end connected to the node 2). Third, SERIES creates a new wire (called Z6 in the figure) between new nodes 3 and 4. The negative end of the new wire is connected to the first new node 3 and the positive end is connected to the second new node 4. Fourth, SERIES inserts a duplicate (called R7 in the figure) of the original component (including all its component values) between new node 3 and original node 1. The positive end of the duplicate is connected to the original node 1 and its negative end is connected to new node 3.

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) and second new node, 4. Figure 5 shows the results of applying the PSS function to resistor R1 from figure 2. The negative end of the new component is connected to the smaller numbered component of the two components that were originally connected to the negative end of the highlighted component. Since C4 bears a smaller number than C5, new node 3 and new wire Z6 are located between original node 2 and C4. Since C2 bears a smaller number than C3, new node 4 and new wire Z8 are located between original node 1 and C2.

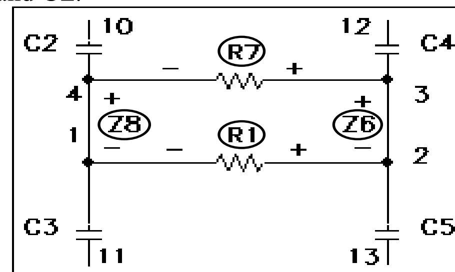

Figure 5 Result of applying `PSS`.

Eight two-argument functions (called VIA0, ..., VIA7) and the two-argument GND ("ground") function enable distant parts of a circuit to be connected together. After execution, writing heads point to two modifiable wires. The VIA functions create a series composition consisting of two wires that each possesses a successor writing head and a numbered port (called a *via*) that possesses no writing head. The port is connected to a designated one of eight imaginary layers (numbered from 0 to 7) of an imaginary silicon wafer. If more than one part of a circuit connects to a

particular numbered layer, all of these parts become electrically connected as if wires were running between them. The two-argument GND function is a special "via" function that establishes a connection directly to ground.

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

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

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

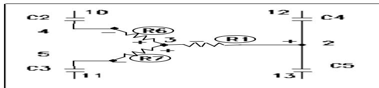

Figure 6 Result of applying the Y1 function.

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 7 illustrates the result of applying the DELTA1 division function to resistor R1 of figure 2 when the active node (node 1) is of degree 3.

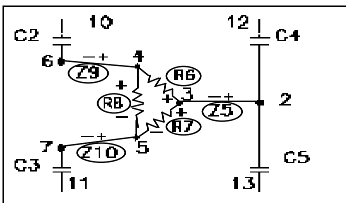

Figure 7 Result of applying DELTA1 function.

8. Preparatory Steps for the Run

8.1. Architecture and Embryonic Circuit

Since this LC filter design problem calls for a one-input, one-output circuit with a source resistor and a load resistor, the embryonic circuit in figure 1 is suitable. Since no automatically defined functions are used in this problem, the architecture of the overall program tree consists of two result-producing branches joined by the connective LIST function. That is, the embryonic circuit initially has two writing heads – one associated with each of the result-producing branches. The terminal set and function sets are the same for both result-producing branches.

8.2. Function and Terminal Sets

The function set, $\mathcal{F}_{aps}$ for an arithmetic-performing subtree associated with each component-creating function contains two two-argument arithmetic functions. That is,

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

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

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

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

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

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

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

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

8.3. Fitness Measure

Note that all of the above could be applied to *any* one-input, one-output LC circuit. It is the fitness measure that directs the evolutionary process (in the search space of circuit-constructing program trees) to a program tree that constructs the desired asymmetric bandpass filter. In other words, circuit structure arises from fitness.

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

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 of its components.

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 10,000 Hz to 200,000 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 relative voltage (in decibels) that is produced by the circuit at the probe point VOUT and the target value for relative voltage. The smaller the value of fitness, the better. A fitness of zero represents an ideal filter. Specifically, the standardized fitness is

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

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

First consider the 87 sample points outside the desired passband. If the relative voltage is below –120 dB for a particular fitness case outside the desired passband, then the deviation is deemed to be zero for that fitness case. If the relative voltage is in the acceptable region (light shading in figures 11, 12, and 13) for a particular fitness case outside the desired passband, the absolute value of the deviation from –120 dB is weighted by 1 for that fitness case. If the relative voltage is in the unacceptable region (white in figures 11, 12, and 13) for a particular fitness case outside the desired passband, the absolute value of the deviation from –120 dB is weighted by 10 for that fitness case. This arrangement reflects the fact that the ideal relative voltage outside the desired passband is –120 dB or less, the fact that the previously described deviations are acceptable, and the fact that other deviations are unacceptable.

Next consider the 14 points inside the desired passband. If the relative voltage is between –0.6 dB and 0.6 dB, the absolute value of the deviation from 0 dB is weighted by a factor of 10.0; however, if the relative voltage lies outside that narrow region, the absolute value of the deviation from 0 dB is weighted by a factor of 100. This arrangement reflects the fact that the ideal relative voltage in the passband is 0 dB, the fact that a ripple of up to 0.6 dB centered at 0 dB is acceptable, and the fact that a relative voltage more than 0.6 dB away from 0 dB in either direction is unacceptable. The factors of 10 and 100 (rather than 1 and 10) were selected to approximately equalize the weight given to the relatively few sample points (i.e., 14) within the desired passband as compared to the relatively large number of sample points (i.e., 87) outside the desired passband.

Hits are defined as the number (0 to 101) of fitness cases for which the relative voltage is acceptable or ideal.

Some of the bizarre circuits that are created by genetic programming cannot be simulated by SPICE. Such circuits are assigned a high penalty value of fitness (10^8).

8.4. Control Parameters

The population size, M, is 640,000. Each subsequent generation of the run is created from the population of the preceding generation by performing 569,600 one-offspring crossover operations (89% of 640,000), 64,000 reproduction operations (10% of 640,000), and 6,400 mutation operations (1% of 640,000). A maximum size of 200 points was established for each of the two result-producing branches. The other minor parameters for controlling the runs were the default values specified in Koza 1994a (appendix D).

8.5. Parallel Implementation

This problem was run on a medium-grained parallel Parsytec computer system consisting of 64 80 MHz PowerPC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer (running Windows). The so-called *distributed genetic algorithm* for parallelization was used with a subpopulation (deme) size of Q = 10,000 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.

9. Results

We made three runs of this problem. Two came close to solving the problem. We now discuss in detail the one run that produced an individual scoring 101 hits.

The best circuit of the 640,000 circuits from generation 0 has a fitness of 59,191, scores 72 hits, and is shown in figure 8. Figure 11 shows its behavior in the frequency domain. The horizontal scale is linear and ranges from 10,000 Hz to 200,000 Hz. The vertical scale is relative voltage in decibels. The curve represents the behavior of this circuit in the frequency domain. As can be seen, the curve in this figure peaks at –14.7 dB when the frequency is 38.40 KHz. This behavior does not even remotely satisfy Nielsen's design goals. However, the narrow peak at 38.40 KHz (which falls far short of the desired height) does, in fact, lie within the desired passband. That is, this inadequate circuit differentially passes frequencies near 38.40 KHz.

About 78% of the programs of generation 0 for this problem produce circuits that cannot be simulated by SPICE. However, the percentage of unsimulatable circuits drops rapidly as new offspring are created by genetic programming using Darwinian selection, crossover, and mutation. The percentage of unsimulatable programs drops to 37% by generation 1. Between generations 2 and 202, the average percentage of unsimulatable circuits is 13.7%. That is, starting as early as the third generation, the vast majority of the circuits are simulatable.

This observation supports the general principle that the individuals in the population in intermediate generations of a run of genetic programming (and random subtrees picked

from them by the crossover operation) differ markedly from the individuals (and their randomly picked subtrees) in the randomly created population of generation 0 of the same run.

Crossover fragments from intermediate generations of a run of genetic programming are very different from the randomly grown subtrees supplied by the mutation operation. This fact is experimental evidence, for this non-trivial problem, that the population serves the vital role in the genetic algorithm of providing a reservoir of useful fragments to rapidly advance the search.

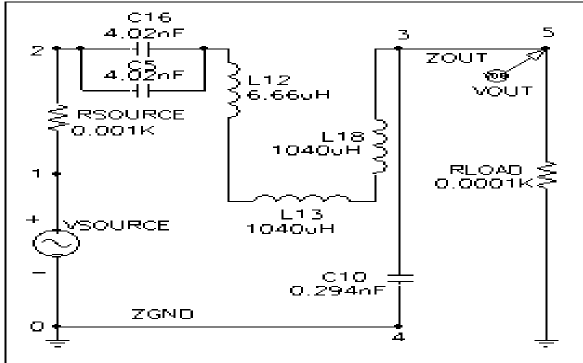

Figure 8 Best circuit of generation 0.

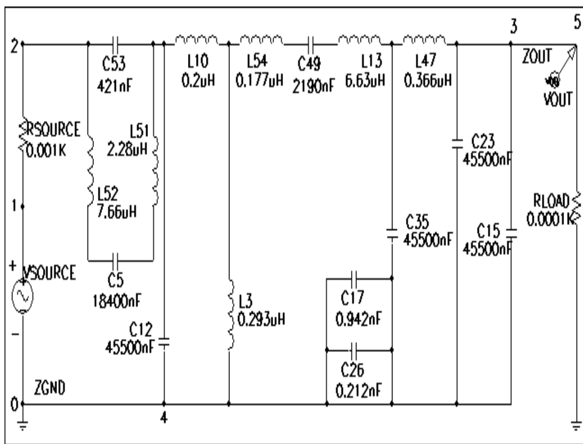

Figure 9 Best circuit of generation 32.

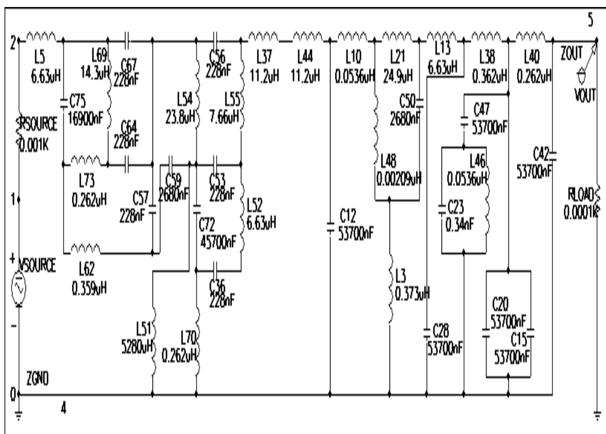

Figure 10 Best-of-run circuit of generation 199.

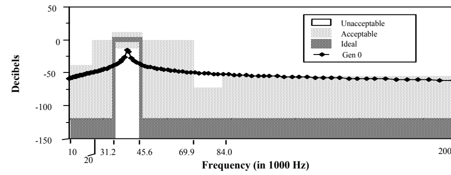

Figure 11 Frequency domain behavior of best circuit of generation 0.

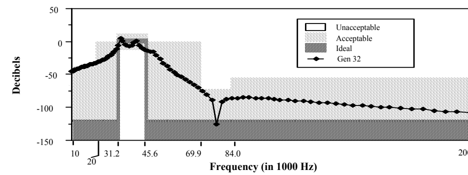

Figure 12 Frequency domain behavior of best circuit of generation 32.

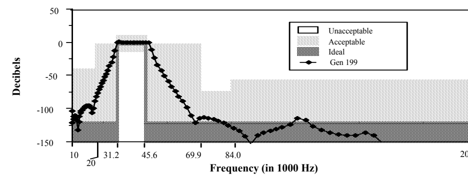

Figure 13 Frequency domain behavior of best-of-run circuit of generation 199.

Point-to-point search techniques (e.g., simulated annealing, hillclimbing) do not have this reservoir from which to draw.

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

search space where parents can successfully sire simulatable offspring by means of the crossover operation.

The best individual program in generation 32 has a fitness of 10,343.0, scores 87 hits, and is shown in figure 9. Figure 12 shows its frequency domain behavior.

The first 100%-compliant program achieving 101 hits appeared in generation 132 and has a fitness of 2252.5. However, we allowed the run to continue. The best-of-run program emerged in generation 199, has a fitness of 2024.0, scores 101 hits, and is shown in figure 10. Figure 13 shows its frequency domain behavior. As can be seen, 100% of the points are in compliance with the design specifications. Moreover, all the points in the passband are within 0.39 dB of 0 dB whereas the specifications merely require a passband ripple of less than 0.6 dB.

10. Related Work

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

10.1. Lowpass "Brick Wall" Filter

Consider the problem of designing a lowpass filter with passband below 1,000 Hz and a stopband above 2,000 Hz (as described in Koza, Bennett, Andre, and Keane 1996a). The voltages in the passband are to be between 970 millivolts and 1 volt (i.e., the passband ripple is 30 millivolts) and the voltages in the stopband between 0 volts and 1 millivolt.

In some runs of this problem, the genetically evolved lowpass filter has a topology that is similar to that employed by human engineers. For example, in one run, the 100% compliant evolved circuit (figure 14) had the recognizable ladder topology of a Butterworth or Chebychev filter (i.e., a composition of inductors horizontally with capacitors as vertical shunts).

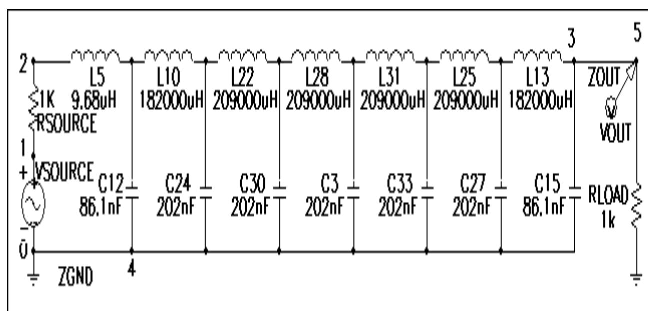

Figure 14 Seven-rung ladder circuit from generation 32.

In another run, a 100%-compliant recognizable "bridged T" arrangement (figure 15) was evolved (involving capacitors C3 and C15 and inductor L11 in conjunction with L14).

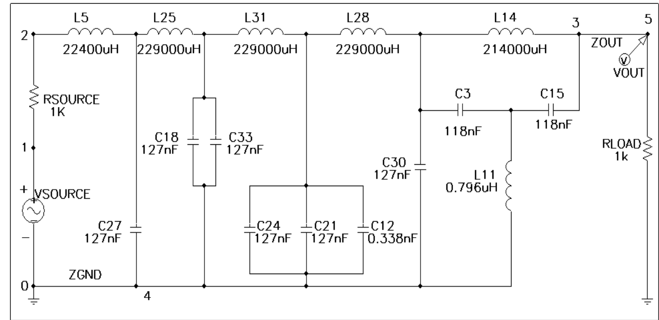

Figure 15 "Bridged T" from generation 64.

Figure 16 shows a 100%-compliant circuit from generation 212 of yet another run. It has a novel topology that no electrical engineer would be likely to create.

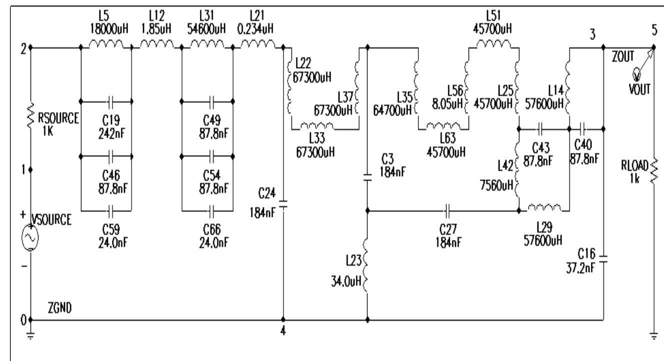

Figure 16 Best circuit from generation 212.

10.2. A 5 dB Amplifier using Transistors

Now consider the problem of designing an amplifier with an amplification factor of 3.5 over the frequency range of 20 Hz to 20,000 Hz. Amplifiers are active circuits; they require active components such as transistors. The major change in the function set needed to evolve the amplifier is to add the resistor-creating R function and the transistor-creating QT functions.

Figure 17 shows a genetically evolved 5 dB amplifier. The boxes highlight a recognizable voltage gain stage and a quasi-Darlington emitter-follower stage.

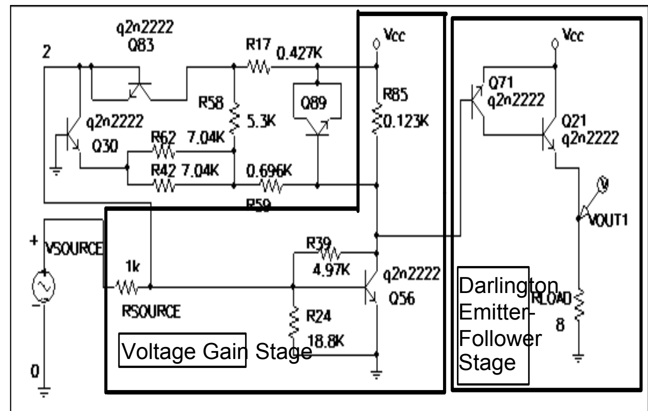

Figure 17 Genetically evolved 5 dB amplifier from generation 45.

10.3. A Crossover Filter

Now consider the problem of designing a crossover (woofer and tweeter) filter with a crossover frequency of 2,512 Hz (as described more fully in Koza, Bennett, Andre, and Keane 1996b). The voltages in the passband are to be between 970 millivolts and 1 volt (i.e., the passband ripple is 30 millivolts) and the voltages in the stopband between 0 volts and 1 millivolt. This problem requires a one-input, two-output embryonic circuit and requires that the fitness be measured at two probe points, but is otherwise similar to the previously discussed circuits. In one run, the best-of-run circuit (figure 18) from generation 137 scores 192 hits. The lowpass part of the best-of-run circuit has the Butterworth topology (but not the Butterworth component values). The highpass part of the best-of-run circuit also has the Butterworth topology (but not the Butterworth component values), except for additional capacitor C36. Note that highpass Butterworth filters have capacitors in series horizontally and inductors as vertical shunts. An analysis of this genetically evolved circuit in the frequency domain indicates that it delivers a response that is slightly better than the combination of lowpass and highpass Butterworth filters of order 7.

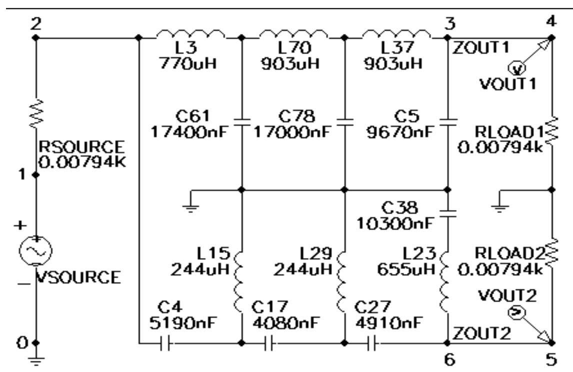

Figure 18 Best circuit from generation 137.

10.4. Related Paper in this Volume

Another paper (appearing next in this volume) on the subject of evolving circuits was presented at this conference.

Acknowledgments

Tom L. Quarles provided helpful advice concerning SPICE. Simon Handley and Scott Brave commented on earlier drafts.

Bibliography

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

Degrauwe, M. 1987. IDAC: An interactive design tool for analog integrated circuits. *IEEE Journal of Solid State Circuits*. 22:1106–1116.

Gruau, Frederic. 1992. *Cellular Encoding of Genetic Neural Networks*. Technical report 92-21. Laboratoire de l'Informatique du Parallélisme. Ecole Normale Supérieure de Lyon. May 1992.

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

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

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

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

Koh, H. Y., Sequin, C. H. and Gray, P. R. 1990. OPASYN: A compiler for MOS operational amplifiers. *IEEE Trans. on Computer Aided Design*. 9:113–125.

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

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

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

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996a. 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 Academic Publishers. 151-170.

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

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

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

Maulik, P. C. Carley, L. R., and Rutenbar, R. A. 1992. A mixed-integer nonlinear programming approach to analog circuit synthesis. *Proceedings of the 29th Design Automation Conference*. Los Alamitos, CA: IEEE Press. Pages 698–703.

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

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

Ning, Z., Kole, M., Mouthaan, T., and Wallings, H. 1992. Analog circuit design automation for performance. *Proceedings of the 14th IEEE CICC*. New York: IEEE Press. Pages 8.2.1–8.2.4.

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

Rutenbar, R. A. 1993. Analog design automation: Where are we? Where are we going? *Proceedings of the l5th IEEE CICC*. New York: IEEE Press. 13.1.1-13.1.8.