Use of Conditional Developmental Operators and Free Variables in Automatically Synthesizing Generalized Circuits using Genetic Programming

John R. Koza
Stanford Medical Informatics
Department of Electrical Engineering
Stanford University
Stanford, California
koza@stanford.edu

Jessen Yu
Genetic Programming Inc.
Los Altos, California
jyu@cs.stanford.edu

Martin A. Keane
Econometrics Inc.
Chicago, Illinois
makeane@ix.netcom.com

William Mydlowec
Genetic Programming Inc.
Los Altos, California
myd@cs.stanford.edu

Abstract

This paper demonstrates that genetic programming can be used to create a circuit-constructing computer program that contains both conditional operations and inputs (free variables). The conditional operations and free variables enable a single genetically evolved program to yield functionally and topologically different electrical circuits. The conditional operations can trigger the execution of alternative sequences of steps based on the particular values of the free variables. The particular values of the free variables can also determine the component value of the circuit's components. Thus, a single evolved computer program can represent the solution to many instances of a problem. This principle is illustrated by evolving a single computer program that yields a lowpass or a highpass filter whose passband and stopband boundaries depend on the program's inputs.

1 Introduction

Genetic algorithms and other techniques of genetic and evolutionary computation are typically used to search for an optimal (or near-optimal) solution to a particular single instance of a problem. For example, an evolutionary algorithm can easily find the numerical values for the real and imaginary parts of the two complex roots of a particular quadratic equation, such as $3x^2 + 4x + 5$. However, a separate run is required to solve each different instance of the problem (e.g., $10x^2 + 2x + 7$).

One of the most important characteristics of computer programs is that they ordinarily contain inputs (free variables) and conditional operations. Free variables enable a single program to produce different outputs based on particular instantiations of the values of its free variables. Conditional operations enable a single program to execute alternative sequences of steps based on the particular instantiations of the values of its free variables. Conditional operations and free variables together potentially enable a single program to solve all instances of a problem (instead of just one instance of the problem). Thus, one computer program containing free variables and conditional operations can be the solution to all equations of the form $ax^2 + bx + c$.

Genetic algorithms and other techniques of genetic and evolutionary computation have been previously used to synthesize electrical circuits, including passive linear circuits composed of two-leaded components (Grimbleby 1995), operational amplifiers (Kruiskamp and Leenaerts 1995), frequency discriminators (Thompson 1996), and other types of circuits, as reported at the various conferences (Sanchez and Tomassini 1996; Higuchi, Iwata, and Liu 1997; Sipper, Mange, and Perez-Uribe 1998; IEEE Computer Society 1999) in the rapidly growing field of evolvable hardware (Higuchi, Niwa, Tanaka, Iba, Hitoshi, de Garis, and Furuya 1993).

Genetic programming (Koza 1992; Koza and Rice 1992; Koza 1994a, 1994b) is capable of automatically creating a computer program to solve a problem. Genetic programming is an extension of the genetic algorithm (Holland 1975). Genetic programming has been shown to be capable of synthesizing the design of both the topology and component values (sizing) for a wide variety of analog electrical circuits from a high-level statement of the circuit's desired behavior and characteristics (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999).

However, all of the previously reported applications of genetic and evolutionary computation have been used to produce only a *single* circuit for a particular *single* high-level statement of the circuit's desired behavior and characteristics.

Since genetic programming searches the space of computer programs, the question arises as to whether genetic programming can be used to create a generalized program containing free variables and conditional operations so that a single genetically evolved program can yield functionally and topologically different circuits. If this were the case, a single program could represent the solution to all instances of a problem (instead of just a single instance of the problem).

Section 2 provides background on genetic programming and the synthesis of electrical circuits by means of developmental genetic programming. Section 3 describes an illustrative problem of analog circuit synthesis for which two functionally and topologically different circuits are required depending on the particular values of two free variables. Section 4 describes the preparatory steps necessary to apply genetic programming to the illustrative problem. Section 5 presents the results.

2 Background

Genetic programming starts with a primordial ooze of thousands of randomly created computer programs and uses the Darwinian principle of natural selection, recombination (crossover), mutation, gene duplication, gene deletion, and certain mechanisms of developmental biology to breed a population of programs over a series of generations.

The *topology* of a circuit involves specification of the gross number of components in the circuit, the identity of each component (e.g., capacitor), and the connections between each lead of each component.

Sizing involves the specification of the values (typically numerical) of each component.

Gruau (1992) demonstrated that the architecture of neural networks can be automatically created by genetic programming by means of a developmental process (cellular encoding). This developmental approach can be extended so as to permit the automatic creation of both the topology and sizing of an electrical circuit by means of genetic programming (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999). When applied to electrical circuits, this developmental approach entails the execution of a circuit-constructing program tree that contains various component-creating, topology-modifying, and development-controlling functions and terminals. The developmental process starts with an initial circuit consisting of an embryo and a test fixture. The test fixture is a fixed (hard-wired) substructure composed of nonmodifiable wires and nonmodifiable electrical components. The test fixture provides access to the circuit's external input(s) and permits probing of the circuit's output. The embryo contains at least one modifiable wire. All development originates from the embryo's modifiable wire(s). The developmental process transforms the initial circuit into a fully developed electrical circuit in the manner specified by the functions and terminals in the circuit-constructing program tree.

Some components of an electrical circuit (e.g., capacitors, inductors, resistors) require specification of their component value(s). The value of each component in a circuit created by genetic programming may be established by an arithmetic-performing subtree associated with its component-creating function. If an arithmetic-performing subtree happens to contain a free variable, it is then a general mathematical expression for determining the component value as a function of the free variable.

Additional information on genetic programming can be found in books such as Banzhaf, Nordin, Keller, and Francone 1998; books such as Langdon 1998, Ryan 1999, and Wong and Leung 2000 in the series on genetic programming from Kluwer Academic Publishers; in edited collections of papers such as the *Advances in Genetic Programming* series of books from the MIT Press (Spector, Langdon, O'Reilly, and Angeline 1999); in the proceedings of the Genetic Programming Conference (Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo 1998); in the proceedings of the Euro-GP conference (Poli, Nordin, Langdon, and Fogarty 1999); in the proceedings of the Genetic and Evolutionary Computation Conference (Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela, and Smith 1999); at web sites such as `www.genetic-programming.org`; and in the *Genetic Programming and Evolvable Machines* journal (from Kluwer Academic Publishers).

3 Illustrative Problem

A *filter* is a one-input, one-output circuit that 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*). A *lowpass filter* passes all frequencies below a certain specified frequency, but stops all higher frequencies. A *highpass filter* stops all frequencies below a certain specified frequency, but passes all higher frequencies.

The illustrative problem here is to automatically create a circuit-constructing program tree containing two inputs (free variables) as well as conditional operations that produces two functionally different circuits depending on the program's two inputs. Specifically, the inputs to each individual computer program in the population are the boundary of the passband (`F1`) and the boundary of the stopband

(F2). If F1 < F2, then a lowpass filter is desired, whereas if F2 < F1, then a highpass filter is desired.

Both desired filters are to have attenuation of 60 decibels (1,000-to-1).

4 Preparatory Steps

4.1 Initial Circuit

The initial circuit for this illustrative problem has one input and one output. It is shown in figure 25.6 of Koza, Bennett, Andre, and Keane 1999. It consists of an embryo embedded in a test fixture. The embryo consists of two modifiable wires. The test fixture has an incoming signal source VSOURCE, a nonmodifiable 1,000 Ohm source resistor, a voltage probe point VOUT (output of the overall circuit), a nonmodifiable 1,000 Ohm load resistor, and other nonmodifiable wires.

4.2 Program Architecture

The architecture of each circuit-constructing program tree has two result-producing branches. There are no automatically defined functions in any program tree in the initial random population (generation 0). However, in later generations, the architecture-altering operations may insert (and delete) zero-argument automatically defined functions of particular individual program trees in the population. A (generous) maximum of five automatically defined functions was established for each program tree in the population.

4.3 Terminal Set

The numerical value for each capacitor or inductor is established by an arithmetic-performing subtree containing perturbable numerical values, arithmetic operations, and the free variables, F1 and F2, representing the boundary of the passband and stopband, respectively. Arithmetic-performing subtrees may appear in both result-producing branches and any automatically defined functions that may be created during the run by the architecture-altering operations. The terminal set, T_{aps}, for the arithmetic-performing subtrees is

T_{aps} = {$\Re$, F1, F2}.

Here $\Re$ denotes a perturbable numerical value. In the initial random generation (generation 0) of a run, each perturbable numerical value is set, individually and separately, to a random value in a chosen range (from -5.0 and +5.0 here). The value returned by the entire arithmetic-performing subtree is interpreted over a range between 10^{-5} and 10^{5} (in units appropriate for the particular component involved). In later generations, a perturbable numerical value may be changed by adding or subtracting a relatively small number determined probabilistically by a Gaussian probability distribution. The standard deviation of the Gaussian distribution is 1.0 (corresponding to one order of magnitude after the value returned by the entire arithmetic-performing subtree is interpreted). The perturbations are implemented by a genetic operation for mutating the perturbable numerical values. The perturbable numerical values are coded by 30 bits in our system. A constrained syntactic structure maintains one function and terminal set for the arithmetic-performing subtrees and a different function and terminal set (below) for all other parts of the program tree.

This approach to numerical constants differs from the approach used in most of our previous work on circuit synthesis, including Koza, Bennett, Andre, and Keane 1999. This approach has the advantage of changing the numerical parameter values by relatively small amounts. Therefore, the space of possible parameter values is most thoroughly searched in the immediate neighborhood of the value of the current numerical value (which is, by virtue of Darwinian selection, usually part of a relatively fit individual). Our experience (albeit limited) is that this perturbation operation for constants (patterned after the Gaussian mutation operation used in evolution strategies and evolutionary programming) appears to work better than our earlier approach. Since this approach is implemented in runs of genetic programming in which crossover is the predominant operation, this approach retains the usual advantage of the genetic algorithm with crossover, namely the ability to exploit and propagate co-adapted sets of numerical values.

The terminal set, T_{rpb}, for other parts of each result-producing branch is

T_{rpb} = {END, SAFE_CUT}.

Each of the above terminals are described in detail in Koza, Bennett, Andre, and Keane 1999. Briefly, the END terminal is a development-controlling function that ends the developmental process for its particular path through the circuit-constructing program tree. SAFE_CUT is a topology-modifying function that preserves circuit validity while deleting a modifiable wire or component from the developing circuit.

The terminal set, T_{adf}, for each automatically defined function (other than their arithmetic-performing subtrees) is

T_{adf} = {END, SAFE_CUT}.

4.4 Function Set

The function set, F_{aps}, for the arithmetic-performing subtrees is

F_{aps} = {IFGTZ_DEVELOPMENTAL, +, -, *, %, REXP, RLOG}.

The three-argument IFGTZ_DEVELOPMENTAL operator executes its second (developmental)

argument (but not its third argument) if its first (numerical argument) is greater than zero; otherwise, this operator executes its third (developmental) argument (but not its second argument).

The two-argument +, -, *, % functions add, subtract, multiply, or divide, respectively, their first argument by their second argument. The one-argument REXP function is the exponential function and the one-argument RLOG function is the natural logarithm of the absolute value. All of these functions are protected in the sense that if the value returned by any of these functions would be less than 10^{-9} or greater than 10^{9}, a value of 10^{-9} or 10^{9}, respectively, is returned.

The function set, F_{rpb}, for all other parts of each result-producing branch is

F_{rpb} = {L, C, SERIES, PARALLEL0, FLIP, NOP, PAIR_CONNECT_0, PAIR_CONNECT_1, THREE_GROUND_0, THREE_GROUND_1, ADF0, ADF1, ADF2, ADF3, ADF4}.

Briefly, the L and C functions are component-creating functions that insert an inductor or capacitor (respectively) into a developing circuit and that establish the numerical value for the inserted component. The SERIES and PARALLEL0 functions modify the topology of the developing circuit by performing a series or parallel division (respectively). The FLIP function reverses the polarity of a component or wire. The NOP (No operation) function is a development-controlling function. The two PAIR_CONNECT functions provide a way to connect two (possibly distant) points in the developing circuit. The two three-argument THREE_GROUND functions each create a via to ground. See Koza, Bennett, Andre, and Keane 1999 for details.

ADF0, … , ADF4 denote automatically defined functions (subroutines). A particular automatically defined function is present in a particular individual in the population at a particular generation during the run only if it has been added (and not deleted) by the architecture-altering operations. The function set, F_{adf}, for each automatically defined function (other than their arithmetic-performing subtrees) consists of F_{rpb} along with whatever automatically defined functions that it is able to call hierarchically.

4.5 Fitness Measure

Genetic programming is a probabilistic search algorithm that searches the space of compositions of the available functions and terminals under the guidance of a fitness measure. The fitness measure is a mathematical implementation of the high-level requirements of the problem and is couched in terms of "what needs to be done" — not "how to do it." The fitness measure may incorporate any measurable, observable, or calculable behavior or characteristic or combination of behaviors or characteristics. Construction of the fitness measure requires translating the high-level requirements of the problem into a mathematically precise computation.

The evaluation of each individual circuit-constructing program tree in the population begins with its execution. The execution progressively applies the functions in the program tree to the embryo of the initial circuit (and to successor circuits during the developmental process), thereby eventually yielding a fully developed circuit. A netlist is created that identifies each component of the fully developed circuit, the nodes to which each component is connected, and the numerical value of each component. The netlist becomes the input to our modified version of the SPICE (Simulation Program with Integrated Circuit Emphasis) program (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994). SPICE then determines the circuit's frequency-domain behavior for probe point VOUT.

The boundary of the passband for a lowpass filter (F1) may range over nine equally spaced values (on a logarithmic scale) between 1,000 Hz and 100,000 Hz, namely 1,000, 1,780, 3,160, 5,620, 10,000, 17,800, 31,600, 56,200, and 100,000 Hz and the corresponding boundary of the stopband of the lowpass filter (F2) will then be 2,000, 3,560, 6,320, 11,240, 20,000, 35,600, 63,200, 112,400, and 200,000 Hz, respectively.

The boundary of the stopband for a highpass filter (F2) may range over the values of 1,000, 1,780, 3,160, 5,620, 10,000, 17,800, 31,600, 56,200, and 100,000 Hz and the corresponding boundary of the passband of the highpass filter (F1) will then be 2,000, 3,560, 6,320, 11,240, 20,000, 35,600, 63,200, 112,400, and 200,000 Hz, respectively.

For each of these 18 combinations of values of the two free variables, F1 and F2, SPICE is instructed to perform an AC small signal analysis and report the circuit's behavior over five decades with each decade being divided into 20 parts (using a logarithmic scale), so that there are a total of 101 sampled frequencies (fitness cases) for each of the 18 combinations of values. The starting frequency for each AC sweep is 100 Hz and the ending frequency is 1,000,000 Hz.

The desired lowpass filter has a passband ending at F1 and a stopband beginning at 2*F1. There should be a sharp drop-off from 1 to 0 Volts in its transitional ("don't care") region between F1 and 2*F1. The desired highpass filter has a stopband ending at F2 and a passband beginning at 2*F2. There should be a sharp rise from 0 to 1 Volt in its transitional ("don't care") region between F2 and 2*F2.

The ideal voltage in the desired passband is 1 volt and the ideal voltage in the desired stopband is 0

volts. A voltage in the desired passband of between 970 millivolts and 1 volt (i.e., a passband ripple of 30 millivolts or less) and a voltage in the desired 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 desired passband and any voltage above 1 millivolts in the desired stopband is unacceptable.

Fitness is the sum, over all 18 combinations of values of the free variables, F1 and F2, and over all 101 fitness cases associated with each combination, 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 (0 or 1 volts). Specifically,

$$F(t) = \sum_{k=1}^{2} \sum_{j=1}^{9} \sum_{i=0}^{100} (W(d(f_i),f_i)d(f_i))$$

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. A smaller value of fitness is better.

The fitness measure is designed to not penalize ideal voltage values, to slightly penalize every acceptable voltage deviation, and to heavily penalize every unacceptable voltage deviation. Specifically, for each of the points in the intended passband, 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. However, 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 points 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 "don't care" point, the deviation is deemed to be zero.

The number of hits is defined as the number of fitness cases (0 to 1,818) for which the voltage is acceptable or ideal or that lie in the "don't care" band.

The SPICE simulator is remarkably robust; however, it cannot simulate every conceivable circuit. In particular, many circuits that are randomly created for the initial population of a run of genetic programming and many circuits that are created by the crossover and mutation operations in later generations are so pathological that SPICE cannot simulate them. Circuits that cannot be simulated receive a penalty value of fitness (10^8) and become the worst-of-generation programs for each generation.

4.6 Control Parameters

The population size, M, was 10,000,000. A (generous) maximum size of 300 points (i.e., total number of functions and terminals) was established for each result-producing branch and a (generous) maximum size of 100 points was established for each automatically defined function. The percentages of the genetic operations for each generation on and after generation 5 are 57.5% one-offspring crossover on internal points of the program tree other than numerical constant terminals, 10% one-offspring crossover on points (internal and external) of the program tree other than numerical constant terminals, 1% mutation on points of the program tree other than numerical constant terminals, 20% mutation on numerical constant terminals, 9% reproduction, 1% subroutine creation, 1% subroutine duplication, and 0.5% subroutine deletion. Since all the programs in generation 0 have a minimalist architecture consisting of just two result-producing branches, we accelerate the appearance of automatically defined functions in the population by using an increased percentage for the architecture-altering operations that add subroutines (i.e., subroutine creation and subroutine duplication) prior to generation 5. Specifically, the percentages for the genetic operations for each generation up to generation 5 are 49%, 10%, 1%, 20%, 5%, 5%, and 1%, respectively. The other parameters are the same default values that we have used previously on a broad range of problems (Koza, Bennett, Andre, Keane 1999).

4.7 Termination

The run was terminated when an individual with 1,818 hits was discovered. This individual was harvested and designated as the result of the run

4.8 Parallel Implementation

This problem was run on a home-built Beowulf-style (Sterling, Salmon, Becker, and Savarese 1999; Bennett, Koza, Shipman, and Stiffelman 1999) parallel cluster computer system consisting of 1,000 350 MHz Pentium II processors (each accompanied by 64 megabytes of RAM). The system has a 350 MHz Pentium II computer as host. The processing nodes are connected with a 100 megabit-per-second Ethernet. The processing nodes and the host use the Linux operating system. The distributed genetic algorithm with unsynchronized generations and semi-isolated subpopulations was used with a subpopulation size of Q = 10,000 at each of D = 1,000 demes. Two processors are housed in each of the 500 physical boxes of the system. As each processor (asynchronously) completes a generation, four boatloads of emigrants from each subpopulation (selected probabilistically based on fitness) are dispatched to each of the four toroidally adjacent

processors. The migration rate is 2% (but 10% if the toroidally adjacent node is in the same physical box).

5 Results

The best individual in generation 0 has a fitness of 3,332.3 and scores 803 hits (out of 1818).

The first of the two result-producing branches from a pace-setting individual from generation 1 (with a fitness of 3,186 and 792 hits) contains the first glimmerings of the problem's ultimate solution.

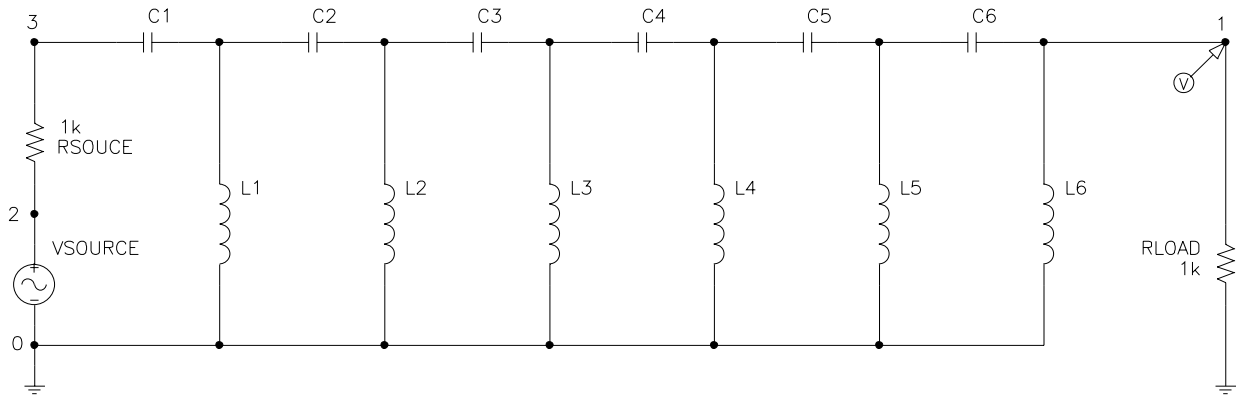

Figure 1 Best-of-run circuit when inputs call for a highpass filter.

Table 1 Component values for highpass filter

Start of transition band	C1	C2 and C4	C3, C5, and C6	L1, L2, L3, L4, and L5	L6
1,000	100.0	57.2	49.9	56,300	100,000
1,778	56.2	32.1	28.1	31,700	63,300
3,162	31.6	18.1	15.8	17,800	35,600
5,623	17.8	10.2	8.88	10,000	20,000
10,000	10.0	5.72	4.99	5,630	11,300
17,782	5.62	3.21	2.81	3,170	6,330
31,622	3.16	1.81	1.58	1,780	3,560
56,234	1.78	1.02	0.888	1,000	2,000
100,000	1.0	0.572	0.499	563	1,130

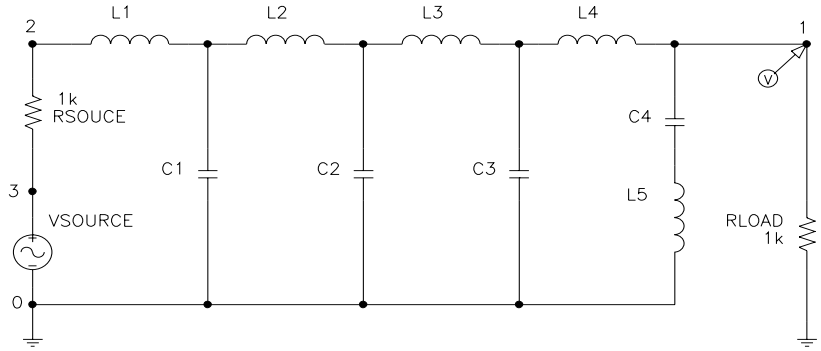

Figure 2 Best-of-run circuit when inputs call for a lowpass filter.

Table 2 Component values for a lowpass filter

Start of transition band	L1	L2, L3, and L4	L5	C1	C2 and C3	C4
1,000	100,000	200,000	58,900	183	219	91.7
1,778	63,400	123,000	33,100	103	123	51.6
3,162	35,600	69,000	18,600	58	69.2	29.0
5,623	20,000	38,800	10,500	32.6	38.9	16.3
10,000	11,300	21,800	5,890	18.3	21.9	9.17
17,782	6,340	12,300	3,310	10.3	12.3	5.16
31,622	3,560	6,900	1,860	5.8	6.92	2.90
56,234	2,000	3,880	1,050	3.26	3.89	1.63
100,000	1,130	2,180	589	1.83	2.19	0.917

```
(IFGTZ_DEVELOPMENTAL
  (*(% F1 F1)
    (- F2 F1))
  (C (RLOG F1)
     (PARALLEL0
       (PARALLEL0 END END END END)
       END
       END
       END))
  (IFGTZ_DEVELOPMENTAL
    (RLOG F2)
    (L F2 END)
    (L F1 END)
  )
)
```

This branch examines the difference between F2 and F1 (after superfluously multiplying the difference by the quotient of two identical quantities). If F1 < F2 (i.e., a lowpass filter is desired), a capacitor is inserted as a shunt. If F2 < F1 (i.e., a highpass filter is desired), an inductor is inserted. This approach is a reasonable ingredient of the ultimate solution.

The best-of-run individual emerged in generation 93. The circuit developed from this best-of-run individual has a near-zero fitness of 0.63568, scores 1,818 hits (i.e., is 100% compliant with the problem's requirements). The result-producing branches of this best-of-run individual have 284 and 282 points (i.e., functions and terminals), respectively. There are three automatically defined functions with 1, 92, and 25 points, respectively. The first result-producing branch calls all three automatically defined functions, while the second result-producing branch calls ADF0 and ADF2. ADF0 hierarchically calls ADF2.

Figure 1 shows the best-of-run circuit from generation 93 that develops when the inputs to the program tree call for a highpass filter (i.e., F1 > F2).

Figure 3 shows the behavior in the frequency domain of the best-of-run circuit from generation 93 that develops when the inputs to the program tree call for a highpass filter for the nine frequencies. The horizontal axis of the figure represents the frequency of the incoming signal and ranges logarithmically over the five decades of frequency between 100 Hz and 1,000,000 Hz. The vertical axis represents the peak voltage of the circuit's output signal and ranges linearly between 0 and +1.2 Volts. As can be seen, the circuit is a 100% compliant highpass filter for all nine frequencies. In particular, for frequencies below F2, the amplitude of the voltages are all near 0.0 volts (i.e., between 0 millivolts and 1 millivolt), as required by the design specifications of the desired highpass filter. Also note that, for frequencies above F1, the amplitude of the voltages are all near 1.0 volts (i.e., between 970 millivolts and 1 Volt).

Table 1 shows the values for the circuit's 12 components (six inductors and six capacitors) when the program's inputs call for a highpass filter. Frequencies are in Hertz; inductors are in micro-Henrys; and capacitors are in nano-Farads. Several of the components have identical values, so the table shows only five distinct values for the highpass filter.

Figure 2 shows the circuit that develops when the inputs to the circuit-constructing program tree call for a lowpass filter.

Figure 4 shows the behavior in the frequency domain of this circuit for the nine frequencies used as the start of the transition band. As can be seen, the circuit is a 100% compliant lowpass filter for each of the nine frequencies.

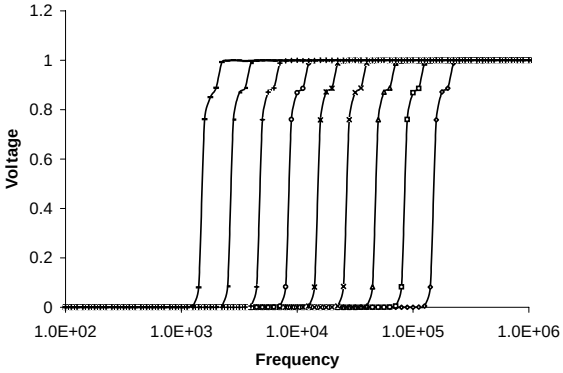

Figure 3 Frequency domain behavior of best-of-run circuit when inputs call for a highpass filter.

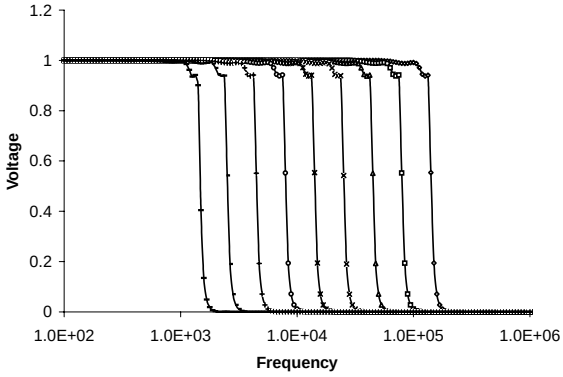

Figure 4 Frequency domain behavior of best-of-run circuit when inputs call for a lowpass filter.

Table 2 shows the values for the circuit's nine components (five inductors and four capacitors) when the program's inputs call for a lowpass filter. Several of the components have identical values, so the table shows only six distinct values for the highpass filter.

Figure 5 shows component values in table 1 (for the highpass filter) for the five components with distinctly values. Both axes use a logarithmic scale.

As can be seen from figure 5 for the parameterized highpass filter, the component values for all five distinct components for all nine frequencies lie along straight lines. Thus, the following formulae (all of which have the frequency F1 in the denominator)

represent the relationship between the component value and frequency for the highpass filter.

$$C1 = \frac{100{,}000nF}{F1} = \frac{100\mu F}{F1}$$

$$C2, C4 = \frac{57{,}200nF}{F1} = \frac{57.2\mu F}{F1}$$

$$C3, C5, C6 = \frac{49{,}900nF}{F1} = \frac{49.9\mu F}{F1}$$

$$L1, L2, L3, L4, L5 = \frac{56{,}300{,}000\mu H}{F1} = \frac{56.3H}{F1}$$

$$L6 = \frac{113{,}000{,}000\mu H}{F1} = \frac{113H}{F1}$$

This inverse proportionality is exactly what you would expect as a means to make a filter scalable over a wide range of frequencies when the impedance (i.e., the load and source resistance here) is fixed (Van Valkenburg 1982).

The equation above yields a value of 113,000 μH for `L6` for `F1`= 1,000Hz, whereas table 1 shows a value of 100,000 μH. This slight discrepancy for `L6` is an artifact of our limiting of component values in the final circuit to a prespecified range.

Six plots for the parameterized lowpass filter (not shown here, but similar to those of figure 5) reveal that the component values for the six distinct components for all nine frequencies also lie along straight lines. The following formulae (all of which have the frequency `F1` in the denominator) represent the relationship between the component value and frequency in the parameterized lowpass filter.

$$L1 = \frac{113{,}000{,}000\mu H}{F1} = \frac{113H}{F1}$$

$$L2, L3, L4 = \frac{218{,}000{,}000\mu H}{F1} = \frac{218H}{F1}$$

$$L5 = \frac{58{,}900{,}000\mu H}{F1} = \frac{58.9H}{F1}$$

$$C1 = \frac{183{,}000nF}{F1} = \frac{183\mu F}{F1}$$

$$C2, C3 = \frac{219{,}000nF}{F1} = \frac{219\mu F}{F1}$$

$$C4 = \frac{91{,}700nF}{F1} = \frac{91.7\mu F}{F1}$$

5.1 Cross Validation

The question arises as to how well the genetically evolved parameterized filter from generation 93 generalizes to previously unseen values of `F1` and `F2`. The evolutionary process is driven by fitness as measured by the above particular set of 18 combinations of in-sample values of frequency `F1` and `F2`. The possibility exists that the circuit that is evolved with a small finite number of values of the frequency `F1` and `F2` (i.e., the so-called "in-sample" or "training" cases) will be overly specialized to those particular values and will prove to be inapplicable to other unseen values of `F1` and `F2`.

Figure 6 shows the behavior, in the frequency domain, of the genetically evolved parameterized filter from generation 93 when inputs call for a highpass filter for four out-of-sample values of frequency `F1` and `F2`. Two of the values of `F1` are from inside the range between 1,000 Hz and 100,000 Hz. They are 4,640 Hz (the antilog of 3 2/3) and 21,540 Hz (the antilog of 4 1/3). Two others are from outside the range, namely 900 Hz and 110,000 Hz. This best-of-run filter from generation 93 is 100% compliant for these four out-of-sample frequencies.

Figure 7 shows the behavior, in the frequency domain, of the genetically evolved parameterized filter from generation 93 when inputs call for a lowpass filter for the same four out-of-sample values of frequency `F1` and `F2`. Again, the genetically evolved parameterized filter is 100% compliant for these four out-of-sample frequencies.

5.2 Computer Time

As one would expect, automatically creating a generalized formula requires considerably more computer time than solving one particular instance of the problem. We made one (and only one) run of the illustrative problem. The best-of-run individual from generation 93 was produced after evaluating 9.4×10^8 individuals (10,000,000 times 94). This required 84.03 hours (3.0252×10^5 seconds) on our 1,000-node parallel computer system — i.e., the expenditure of 1.05882×10^{17} computer cycles (about 105 peta-cycles of computer time).

6 Conclusion

This paper demonstrated that genetic programming can automatically create a single two-input circuit-constructing computer program that contains both conditional operations and inputs (free variables) and that produces either a lowpass or highpass filter with specified boundaries for its passband and stopband. The single evolved computer program represents the solution to multiple instances of the illustrative problem. The ability of a single evolved program to represent the solution to multiple instances of a problem depended on the fact that the evolutionary search was conducted in the space of computer programs containing conditional operations and free variables.

7 Future Work

The work reported in this paper is part of ongoing work aimed at establishing the principle that genetic programming can be used to automatically create generalized, parameterized programs to solve multiple instances of difficult problems. For example, we have recently reported on the automatic synthesis of both the topology and tuning of a generalized, parameterized controller for two industrially useful families of plants using genetic programming (Keane, Yu, and Koza 2000). We are currently working on similar problems in additional domains.

References

Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E. (editors). 1999. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA*. San Francisco, CA: Morgan Kaufmann.

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1998. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C. 1998. *Genetic Programming: First European Workshop. EuroGP'98. Paris, France, April 1998 Proceedings. Paris, France. April l998*. Lecture Notes in Computer Science. Volume 1391. Berlin, Germany: Springer-Verlag.

Bennett, Forrest H III, Koza, John R., Shipman, James, and Stiffelman, Oscar. 1999. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E. (editors). *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida*. San Francisco, CA: Morgan Kaufmann. 1484-1490.

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 (GALESIA)*. 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.

Higuchi, Tetsuya, Iwata, Masaya, and Lui, Weixin (editors). 1997. *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*. Lecture Notes in Computer Science, Volume 1259.

Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. 1993. Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. Evolving hardware with genetic learning: A first step towards building a Darwin machine. In Meyer, Jean-Arcady, Roitblat, Herbert L. and Wilson, Stewart W. (editors). *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: The MIT Press. 1993. Pages 417 – 424.

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

IEEE Computer Society. 1999. *Proceedings of the First NASA / DOD Workshop on Evolvable Hardware, Pasadena, California, July 19 - 21, 1999*. Los Alamitos, CA. IEEE Computer Society.

Keane, Martin A., Yu, Jessen, and Koza, John R. 2000. Automatic synthesis of both the topology and tuning of a common parameterized controller for two families of plants using genetic programming. *GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference, July 8-12, 2000, Las Vegas, Nevada, USA*. San Francisco, CA: Morgan Kaufmann.

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., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). 1998. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A., and Brave Scott. 1999. *Genetic Programming III Videotape: Human-Competitive Machine Intelligence*. San Francisco, CA: Morgan Kaufmann.

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

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

Langdon, William B. 1998. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Amsterdam: Kluwer.

Poli, Riccardo, Nordin, Peter, Langdon, William B., and Fogarty, T. C. 1999. *Genetic Programming: Second European Workshop. EuroGP'99. Proceedings*. Lecture Notes in Computer Science. Volume 1598. Berlin: Springer-Verlag.

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, Univ. of California. Berkeley, CA. March 1994.

Ryan, Conor. 1999. *Automatic Re-engineering of Software Using Genetic Programming*. Amsterdam: Kluwer Academic Publishers.

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

Sipper, Moshe, Mange, Daniel, and Perez-Uribe. 1998. *Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98, Lausanne, Switzerland, September 1998 Proceedings*. Lecture Notes in Computer Science 1478. Berlin: Springer-Verlag.

Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter (editors). 1999. *Advances in Genetic Programming 3*. Cambridge, MA: MIT Press.

Sterling, Thomas L., Salmon, John, Becker, D. J., and Savarese, D. F. 1999. *How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters*. Cambridge, MA: MIT Press.

Thompson, Adrian. 1996a. Silicon evolution. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. 444–452.

Van Valkenburg, M. E. 1982. *Analog Filter Design*. Fort Worth, TX: Harcourt Brace Jovanovich.

Wong, Man Leung and Leung, Kwong Sak. 2000. *Data Mining Using Grammar Based Genetic Programming and Applications*. Amsterdam: Kluwer Academic Publishers.

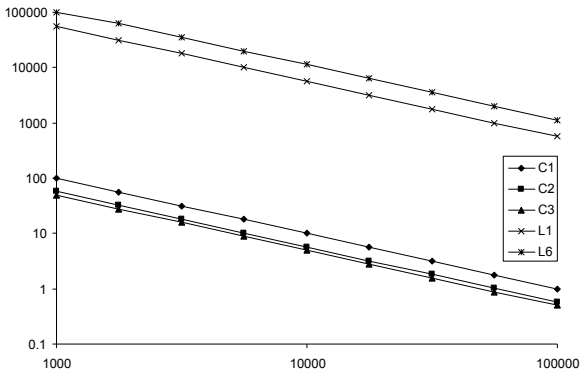

Figure 5 Comparison of component values for highpass filter.

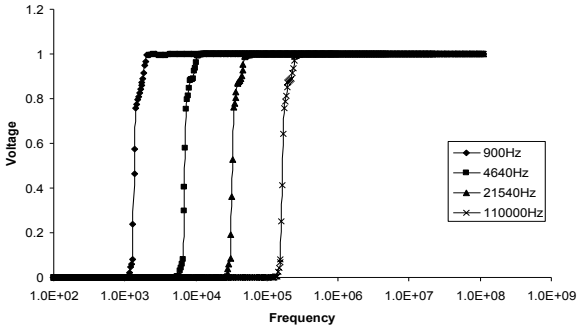

Figure 6 Frequency domain behavior of best-of-run circuit for four out-of-sample values of frequency when inputs call for a highpass filter.

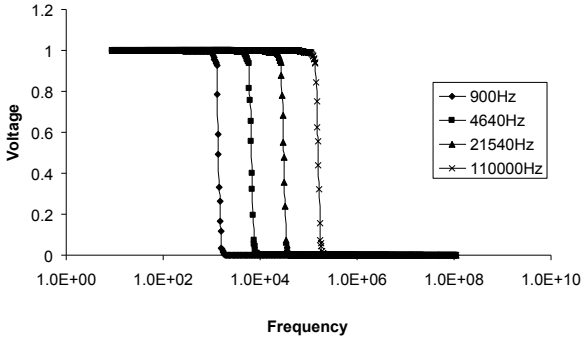

Figure 7 Frequency domain behavior of best-of-run circuit for four out-of-sample values of frequency when inputs call for a lowpass filter.