Automatic Synthesis of Electrical Circuits Containing a Free Variable Using Genetic Programming

John R. Koza
Stanford University
Los Altos, CA 94023
koza@stanford.edu

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

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

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

Abstract

A mathematical formula containing one or more free variables is "general" in the sense that it represents the solution to all instances of a problem (instead of just the solution of a single instance of the problem). For example, the familiar formula for solving a quadratic equation contains free variables representing the coefficients of the to-be-solved equation. This paper demonstrates, using an illustrative problem, that genetic programming can automatically create the design for both the topology and component values for an analog electrical circuit in which the value of each component in the evolved circuit is specified by a mathematical expression containing a free variable. That is, genetic programming is used to evolve a general parameterized circuit that satisfies the problem's high-level requirements. The evolved circuit has been cross-validated on unseen values of the free variable.

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. Although 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 $3x^2 + 4x + 5$, 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 the particular values of its free variables. Conditional operations enable a single program to execute alternative sequences of steps based on the particular values of the 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, a computer program containing free variables and conditional operations has the ability to solve all quadratic equations.

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; 1994a, 1994b) is a technique for automatically creating computer programs to solve, or approximately solve, problems. 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 evolved individual could represent the solution to all instances of a problem (instead of just a single instance of the problem).

This paper addresses the question of whether genetic programming can create the design for both the topology and component values for an analog electrical circuit in which the value of each component in the evolved circuit is specified by a mathematical expression containing a free variable.

Section 2 provides sources on genetic programming and briefly reviews automatic synthesis of electrical circuits by means of developmental genetic programming. Section 3 describes an illustrative problem. Section 4 itemizes the preparatory steps necessary to apply genetic programming to the illustrative problem. Section 5 presents the results.

2 Automatic Circuit Synthesis using Developmental Genetic Programming

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.

Both the topology and sizing of an electrical circuit can be created by genetic programming by means of a developmental process (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999). This developmental process entails the execution of a circuit-constructing program tree that contains various component-creating, topology-modifying, and development-controlling functions. An initial circuit consisting of an embryo and a test fixture is the starting point of the developmental process for transforming a program tree in the population into a fully developed electrical circuit. The embryo contains at least one modifiable wire. 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. A test fixture has ports that enable an embryo to be embedded into it. An embryo has one or more ports that enable it to communicate with the test fixture. All development originates from the modifiable wires.

Many components appearing in electrical circuit (e.g., capacitors, inductors) are specified by component values. 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 contains 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 electronic 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.

The problem is to evolve a general analog electrical circuit for a lowpass filter whose passband ends at frequency f and whose stopband starts at frequency $2f$ with 60 decibels (1,000-to-1) of attenuation. (A *decibel* is a unitless measure of relative voltage that is defined as 20 times the common logarithm of the ratio between the two voltages). The frequencies between f and $2f$ constitute a "don't care" region.

4 Preparatory Steps

Seven major preparatory steps are required to apply genetic programming to a problem of circuit synthesis using developmental genetic programming: (1) identify the initial circuit for the circuit developmental process, (2) determine the architecture of the circuit-constructing program trees, (3) identify the primitive functions of the circuit-constructing program trees, (4) identify the terminals of the circuit-constructing program trees, (5) define the fitness measure, (6) choose control parameters, and (7) determine the termination criterion and method of result designation.

4.1 Initial Circuit

Figure 1 shows a one-input, one-output initial circuit consisting of an embryo embedded in a test fixture. The embryo consists of two modifiable wires Z0, and Z1. The test fixture has an incoming signal source VSOURCE, a 1,000 Ohm source resistor RSOURCE, a nonmodifiable wire ZOUT, a voltage probe point VOUT (the output of the overall circuit), a 1,000 Ohm load resistor LOAD, and a nonmodifiable wire ZGND connecting to ground.

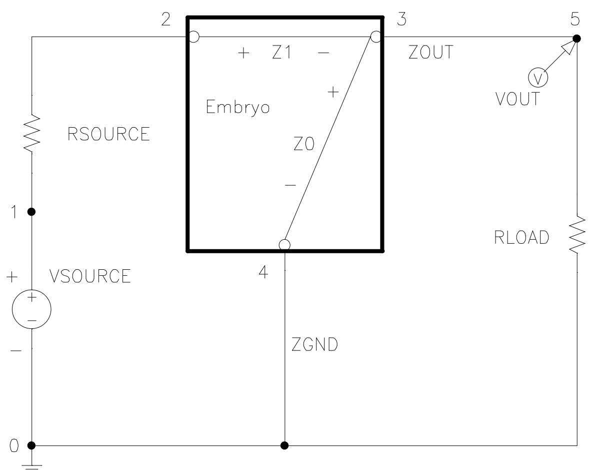

Figure 1 One-input, one-output initial circuit.

4.2 Program Architecture

The architecture of each circuit-constructing program tree has two result-producing branches. There are no automatically defined functions (Koza 1992, 1994a, 1994b) in any program tree in the initial random population (generation 0). However, in later generations, the architecture-altering operations may insert (and delete) one-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 variable *f* representing the frequency of the end of the desired filter's passband. 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 value returned by an entire arithmetic-performing subtree is interpreted as a component value lying in a range of (positive values) between 10^{-5} and 10^{5}. The terminal set, T_{aps}, for the arithmetic-performing subtrees is

$T_{aps} = \{\Re, F\}$.

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). 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 here (i.e., one order of magnitude after the value returned by an 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, ARG0}.

Here ARG0 is the dummy argument (formal parameter) of the automatically defined function.

4.4 Function Set

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

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

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 217,000-line SPICE (Simulation Program with Integrated Circuit Emphasis) simulation program (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994). SPICE is an industrial strength simulator. Hundreds of thousands of copies are in use by practicing electrical engineers throughout the world. SPICE is remarkably robust in general. The frequencies and components involved in this problem are handled without difficulty by SPICE. The simulator then determines the behavior of the circuit.

Since the high-level statement of the behavior for the desired filter circuit is expressed in terms of frequencies, the output voltage VOUT is measured in the frequency domain.

The free variable *f* ranges over nine values that are equally spaced (on a logarithmic scale) in the range between 1,000 Hz and 100,000 Hz. Specifically, the nine values of *f* are 1,000, 1,780, 3,160, 5,620, 10,000, 17,800, 31,600, 56,200, and 100,000 Hz. For each of the nine values for the free variable *f*, 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 value of *f* . The starting frequency for each AC sweep is *f*/1,000 and the ending frequency is 100*f*. The desired lowpass filter has a passband ending at *f* and a stopband beginning at 2 *f* . There should be a sharp drop-off from 1 to 0 Volts in the transitional (“don't care”) region between *f* and 2 *f*. For example, if *f* is 3,160 Hz (the third of the nine fitness cases), then the AC small signal analysis is performed between 3.16 Hz (three decades below *f*) and 316,000 Hz (two decades above *f*) and desired lowpass filter has a passband ending at 3,160 Hz and a stopband beginning at 6,320 Hz.

The ideal voltage in the passband of the desired lowpass filter 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 nine values of the free variable *f* and over all 101 fitness cases associated with each of the nine values of *f* 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). A smaller value of fitness is better. Specifically,

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

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*.

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 of the “don't care” points between *f* and 2 *f*, the deviation is deemed to be zero.

The number of hits is defined as the number of fitness cases (0 to 909) 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. These circuits receive a high 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% one-offspring crossover on internal points of the program tree other than numerical constant terminals, 10% one-offspring crossover on points 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 1% 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%, 9%, 5%, 5%, and 1%, respectively. The other parameters (including details concerning tournament selection and other aspects of the run) 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 909 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. As each processor (asynchronously) completes a generation, four boatloads of emigrants from each subpopulation are dispatched to each of the four toroidally adjacent processors. The 1,000 processors are hierarchically organized. There are $5 \times 5 = 25$ high-level groups (each containing 40 processors). If the adjacent node belongs to a different group, the migration rate is 2% and emigrants are selected based on fitness. If the adjacent node belongs to the same group, emigrants are selected randomly and the migration rate is 5% (10% if the adjacent node is in the same physical box).

5 Results

The fitness of the best individual from generation 0 is 957.1.

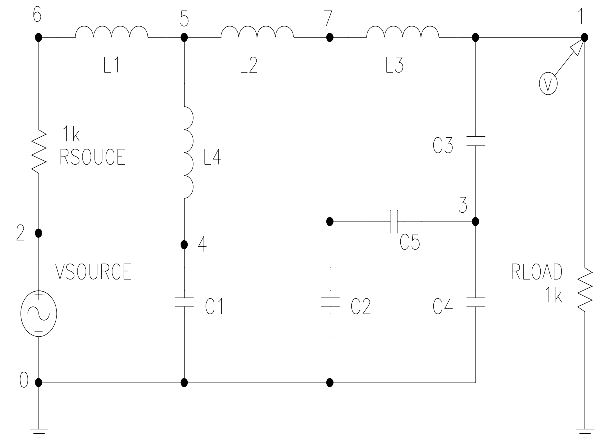

Figure 2 Genetically evolved parameterized filter.

The first individual to achieve 100% compliance with the problem's requirements by scoring 909 hits (out of 909) appeared in generation 78 (figure 2). This best-of-run circuit has a fitness of 0.18450 and scores 909 hits (out of 909). The circuit has nine components (four inductors and five capacitors). The circuit-constructing program tree has two result-producing branches (with 146 and 295 points, respectively) and five automatically defined functions (with 4, 3, 5, 3, and 1 points, respectively). Both result-producing branches refer to `ADF0` once. `ADF0` hierarchically refers to `ADF3` and they together evaluate to 1 - *f*. The other three automatically defined functions are not referenced.

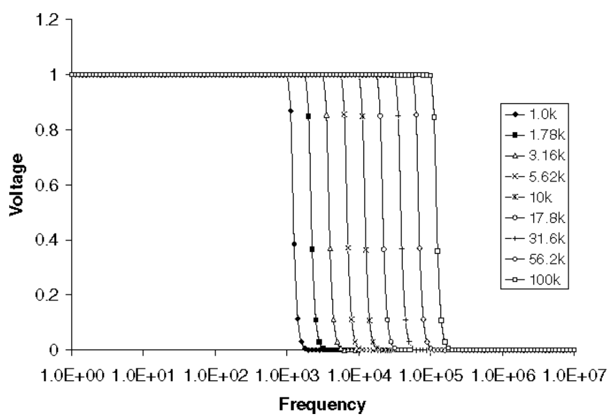

Figure 3 Frequency domain behavior of genetically evolved parameterized filter for nine values of frequency *f*.

Figure 3 shows the behavior, in the frequency domain, of the genetically evolved parameterized filter from generation 78 for each of nine values of the free variable *f*. The horizontal axis represents the frequency of the incoming signal and ranges logarithmically over the seven decades of frequency between 1 Hz and 10 MHz. The 101 equally spaced filled circles (many of which overlap in the figure) along each of the nine curves represent the 101 fitness cases (sampled frequencies). The vertical axis represents the peak voltage of the circuit's output signal and ranges linearly between 0 and +1.2 Volts. The amplitude of the voltages produced by the genetically evolved parameterized filter for frequencies below *f* are all between 970 millivolts and 1 Volt). The amplitude of the voltages produced by the genetically evolved parameterized filter for frequencies above 2*f* are all between 0 millivolts and 1 millivolt. Thus, the evolved parameterized filter is 100% compliant for all sampled frequencies for all nine values of the free variable *f*.

The component values for the circuit's nine components (four inductors and five capacitors) are shown below. Note that each component value is a function of the problem's free variable *f*. Inductors are in micro-Henrys and capacitors are in nano-Farads.

$$L1 = \frac{8.0198\times 10^{7}}{f}$$

$$+\ln f \approx \frac{2.4451\times 10^{8}}{f} + \ln f$$

$$L2 = \frac{1.3406\times 10^{-8}\left(4.7387\times 10^{12} + f\right)\left(1.3331\times 10^{16} + 9.3714\times 10^{5} f + f^{2}\right)}{f\left(3.4636\times 10^{12} + f\right)}$$

$$+\ln f \approx \frac{2.4451\times 10^{8}}{f} + \ln f$$

$$L3 = \frac{2.0262\times 10^{8}}{f} + 2\ln f$$

$$L4 = \frac{3.7297\times 10^{7}}{f}$$

$$C1 = \frac{1.6786\times 10^{5}}{f}$$

$$C2 = \frac{1.6786\times 10^{5}}{f}$$

$$C3 = \frac{1.3552\times 10^{5}}{f}$$

$$C4 = \frac{6.4484\times 10^{5}}{f}$$

$$C5 = \frac{1.1056\times 10^{5}}{f}$$

As can be seen, all the above component values (except for `L2` and `L3`) are inversely proportional to *f*. For `L2` and `L3`, the component values are approximately inversely proportional to *f* for *f* less than 1,000,000 Hz. This 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). Thus, the mathematical expressions for each of the component values in the evolved circuit have a reasonable interpretation in electrical engineering terms.

5.1 Cross Validation

The question arises as to how well the genetically evolved parameterized filter from generation 78 generalizes to previously unseen values of the frequency *f*. The evolutionary process is driven by fitness as measured by the above particular set of nine in-sample values of frequency *f*. The possibility exists that the circuit that is evolved with a small finite number of values of the frequency *f* (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 *f*. Figure 4 shows the behavior, in the frequency domain, of the genetically evolved parameterized filter from generation 78 for four out-of-sample values of frequency *f*. Two of the values of *f* 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 100 Hz and 1,000,000 Hz (each a full decade outside the original range of frequencies). As can be seen, the genetically evolved parameterized filter from generation 78 is 100% compliant for these four out-of-sample values of frequency *f*.

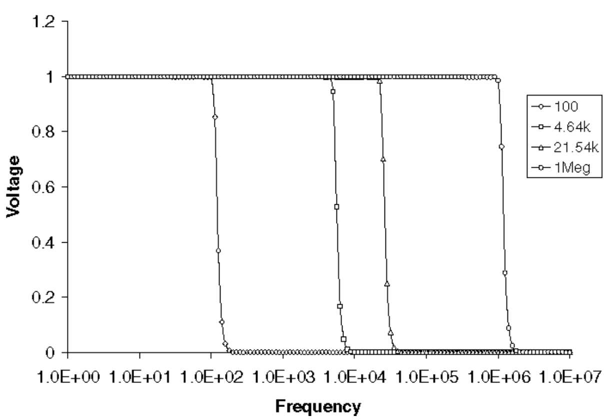

Figure 4 Frequency domain behavior of genetically evolved parameterized filter for four out-of-sample values of frequency *f*.

5.2 Computer Time

As one would expect, automatically creating a generalized formula to solve an entire category of problems requires considerably more computer time than solving one particular instance of the problem. The best-of-run individual from generation 78 was produced after evaluating 7.9×10^8 individuals (10,000,000 times 79). This required 44.9 hours (1.584×10^5 seconds) on our 1,000-node parallel computer system — that is, the expenditure of 5.54×10^{16} computer cycles (about 55 peta-cycles of computer time).

6 Conclusions

This paper demonstrated that genetic programming can automatically create the design for both the topology and component values for an analog electrical circuit in which the value of each component in the evolved circuit is specified by a mathematical expression containing a free variable. That is, genetic programming evolved a circuit that is general in the sense that it represents the solution to all instances of a problem (instead of just the solution of a single instance of the problem). The mathematical expressions for each of the component values in the evolved circuit have a reasonable interpretation in electrical engineering terms. The evolved circuit has been cross-validated on unseen values of the free variable.

References

Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert 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, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert 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. 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.

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*. 2nd. Ed. Cambridge, MA: MIT 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.

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

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, Terence 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, University 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, Donald J., and Savarese, Daniel F. 1999. *How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters*. Cambridge, MA: MIT Press.

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