Use of Time-Domain Simulations in Automatic Synthesis of Computational Circuits Using Genetic Programming

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

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

Abstract

Previously reported applications of genetic programming to the automatic synthesis of computational circuits have employed simulations based on DC sweeps. DC sweeps have the advantage of being considerably less time-consuming than time-domain simulations. However, this type of simulation does not necessarily lead to robust circuits that correctly perform the desired mathematical function over time. This paper addresses the problem of automatically synthesizing computational circuits using multiple time-domain simulations and presents results involving the synthesis of both the topology and sizing for a squaring, square root, and multiplier computational circuit and a lag circuit (from the field of control).

1 Introduction

An analog electrical circuit whose output is a well-known mathematical function (e.g., square, square root) is called a *computational circuit*.

Analog computational circuits are especially useful when the mathematical function must be performed more rapidly than is possible with digital circuitry (e.g., for real-time signal processing at extremely high frequencies). Analog computational circuits are also useful when the need for a single mathematical function in an analog circuit does not warrant converting an analog signal into a digital signal (using an analog-to-digital converter), performing the mathematical function in the digital domain (requiring a general purpose digital processor consisting of millions of transistors), and then converting the result to the analog domain (using a digital-to-analog converter).

The design of computational circuits is exceedingly difficult even for seemingly mundane mathematical functions. Success usually relies on the clever exploitation of some aspect of the underlying device physics of the components (e.g., transistors) that is uniquely suited to the particular desired mathematical function. Because of this, the implementation of each different mathematical function typically requires an entirely different design approach (Gilbert 1968, Sheingold 1976, Babanezhad and Temes 1986).

The *topology* of a circuit involves specification of the gross number of components in the circuit, the identity of each component (e.g., resistor, transistor), and the connections between each lead of each component. *Sizing* involves the specification of the values (typically numerical) of each component possessing a component value (e.g., resistors).

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 is 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). In particular, genetic programming has been previously used to automatically synthesize the design for computational circuits, including squaring, cubing, square root, cube root, logarithm, and Gaussian circuits.

However, all of the previously reported applications of genetic programming to the automatic synthesis of computational circuit have employed simulations based on DC sweeps. DC sweeps are considerably less time-consuming than time-domain simulations. This type of simulation does not necessarily lead to robust circuits that correctly perform the desired mathematical function over time. This paper addresses the problem of automatically synthesizing computational circuits using multiple time-domain simulations.

Section 2 describes automatic synthesis of electrical circuits by means of developmental genetic programming. Section 3 itemizes the preparatory steps necessary to apply genetic programming to the four illustrative problems involving a squaring, square root, and multiplier computational circuit and a lag circuit (from the field of control). Sections 4, 5, 6, and 7 present results for the four problems.

2 Automatic Circuit Synthesis using Developmental Genetic Programming

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. A simple initial circuit is the starting point of the developmental process for creating a fully developed electrical circuit. The initial circuit consists of an embryo and a test fixture. The embryo contains at least one modifiable wire. The test fixture is a fixed (hard-wired) substructure composed of nonmodifiable wire(s) and nonmodifiable electrical component(s). The test fixture provides access to the circuit's external input(s) and permits probing of the circuit's output. All development originates from the modifiable wires. The execution of the component-creating, topology-modifying, and development-controlling functions in the program tree transforms the initial circuit into a fully developed circuit.

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, Banzhaf, Langdon, Miller, Nordin, and Fogarty 2000); 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 Preparatory Steps

3.1 Initial Circuit

A one-input, one-output initial circuit with one modifiable wire (shown in figure 30.1 of Koza, Bennett, Andre, and Keane 1999) was used for the problems involving the squaring circuit and the square root circuit. The initial circuit has an incoming signal source, a 1,000 Ohm source resistor, a voltage probe point VOUT, and a 1 Ohm load resistor.

Since the multiplication function has two inputs, a two-input, one-output initial circuit (shown in figure 29.15 of Koza, Bennett, Andre, and Keane 1999) with three modifiable wires was used for the multiplier.

The lag function applies the transfer function $\frac{1}{(1+\tau s)}$ to a time-domain input signal, where τ is the time constant and s is the Laplace operator. Since the lag function has two inputs (the incoming signal and the variable value of τ), the initial circuit used in the multiplier problem was used for the lag problem.

3.2 Program Architecture

The circuit-constructing program tree has one result-producing branch for each modifiable wire in the embryo of the initial circuit. Thus, each program tree has one result-producing branch for the problems involving the squaring circuit and the square root circuit and three result-producing branches for the problems involving the multiplier circuit and the lag circuit. Automatically defined functions were not used.

3.3 Terminal Set

The value of each component in a circuit possessing a parameter (e.g., capacitors, resistors) is established by an argument of its component-creating function. The argument is 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 (-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 distribution (with a standard deviation of 0.5 here). This numerical value is subsequently interpreted as a component value lying in a range of positive values between 10^{-5} and 10^{5}.

The perturbations are implemented by a genetic operation for mutating the numerical values. In addition, single numerical constants can be swapped by a crossover operation that operates only the numerical constants. The crossover operation may move subtrees containing component-creating functions and associated numerical constants between program trees. The mutation operation may create subtrees containing component-creating functions and their associated numerical constants.

The perturbable numerical values are coded by 32 bits in our system. A constrained syntactic structure maintains the distinction between the numerical values and 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 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, in part, after the Gaussian mutation operation used in evolution strategies and evolutionary programming) appears to work better than our earlier approach. Since crossover is the predominant operation in our runs of genetic programming, this approach retains the usual advantage of the genetic algorithm with crossover, namely the ability to exploit co-adapted sets of functions and numerical values.

Aside from the numerical constants, the terminal set for each result-producing branch is

T_{rpb} = {END, SAFE_CUT}.

These terminals are each described in detail in Koza, Bennett, Andre, and Keane 1999. Briefly, the END terminal is a development-controlling zero-argument function that ends the developmental process for a particular path through the circuit-constructing program tree. SAFE_CUT is a topology-modifying function that deletes a modifiable wire or component from the developing circuit while preserving circuit validity.

3.4 Function Set

The function set for each result-producing branch is

F_{rpb} = {R, C, L, SERIES, PARALLEL0, PARALLEL1,
FLIP, NOP,
RETAINING_THREE_GROUND_0,
RETAINING_THREE_GROUND_1,
RETAINING_THREE_POS5V_0,
RETAINING_THREE_POS5V_1,
PAIR_CONNECT_0, PAIR_CONNECT_1,
Q_DIODE_NPN, Q_DIODE_PNP,
Q_THREE_NPN0, ..., Q_THREE_NPN11,
Q_THREE_PNP0, ..., Q_THREE_PNP11,
Q_POS5V_COLL_NPN,
Q_POS5V_EMIT_PNP,
Q_GND_EMIT_NPN, Q_GND_EMIT_PNP}.

See Koza, Bennett, Andre, and Keane 1999 for a detailed description of each of these functions. Briefly, the R, C, and L functions are component-creating functions that insert a resistor, capacitor, or inductor (respectively) into a developing circuit and that establish the numerical value for the inserted component. The functions beginning with Q insert transistors or diodes into a developing circuit. The SERIES and PARALLEL functions modify the topology of the developing circuit by performing a series or parallel division (respectively). The FLIP function reverses the polarity of a two-leaded component or wire. The NOP (No operation) function is a one-argument development-controlling function. The two PAIR_CONNECT functions provide a way to connect two points in the developing circuit. The two three-argument RETAINING_THREE_GROUND functions each create a via to ground.

3.5 Fitness Measure

The evaluation of each individual circuit-constructing program tree in the population begins with its execution. The execution progressively applies the component-creating, topology-modifying, and development-controlling 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) simulation program (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994). SPICE determines the circuit's behavior in terms of the output voltage VOUT in the time domain. Each individual circuit in the population is exposed to four time-domain signals for a (simulated) duration of one second each. Figures 1, 2, 3, and 4 show the four time-domain signals for the problem of synthesizing the squaring function. These input signals are structured to provide a representative mixture of input values, all of which produce outputs that are well within the range that the transistors can handle (i.e., below 4 volts).

We first considered running this problem using the "obvious" fitness measure consisting of the sum, over a certain number of time steps for each of the four fitness cases, of the absolute value of the difference between the actual output voltage at probe point VOUT and the desired output voltage. However, initial tests of our code on a single Pentium workstation indicated that this fitness measure often yielded circuits having noticeable spikes in the output. These spikes were reminiscent of the glitches that we encountered in previous work involving digital-to-analog converter circuits (Bennett, Koza, Keane, Yu, Mydlowec, and Stiffelman 1999).

Therefore, we adopted the same fitness measure that we used to eliminate the spikes in the earlier work. Specifically, we considered the difference only at times that correspond to SPICE's internally created turn-defining points. Thus, the fitness measure for the computational circuit is the sum, for each of SPICE's turns, of the absolute value of the difference between the actual output voltage at the probe point VOUT and the desired output voltage. Unsimulatable circuits receive a high penalty value of fitness (10^8).

For the square root problem, the input signals necessarily differ from the signals used for the squaring circuit because no negative values are used as inputs.

Each individual circuit is exposed to four time-domain signals for a (simulated) duration of one second each, as shown in figures 5, 6, 7, and 8.

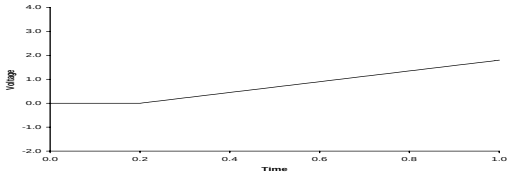


Figure 1 Rising ramp for squaring circuit.

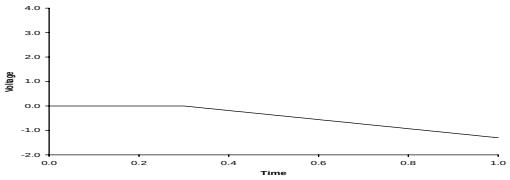


Figure 2 Falling ramp for squaring circuit.

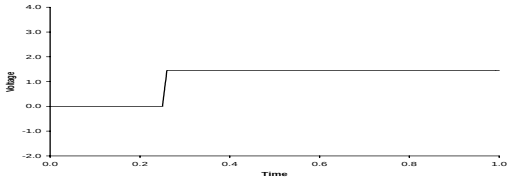


Figure 3 Rising step for squaring circuit.

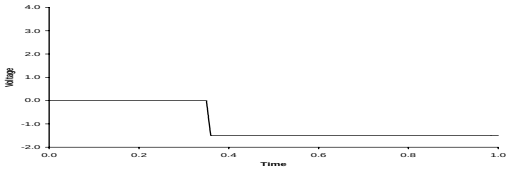


Figure 4 Falling step for squaring circuit.

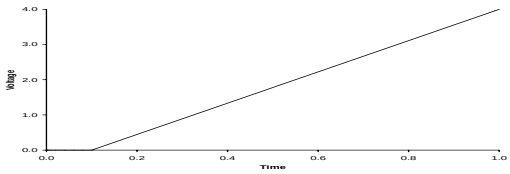


Figure 5 Rising ramp for square root circuit.

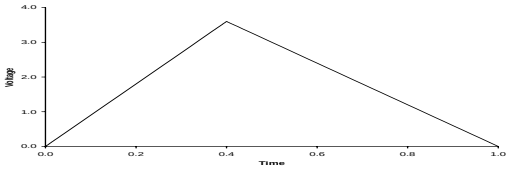


Figure 6 Triangle for square root circuit.

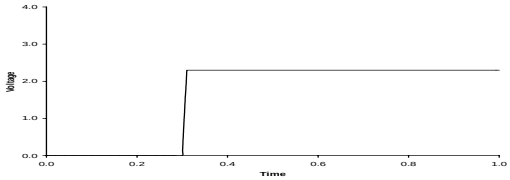


Figure 7 Rising step for square root circuit.

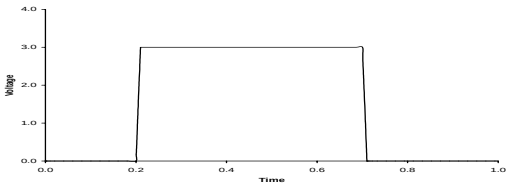


Figure 8 Pulse for square root circuit.

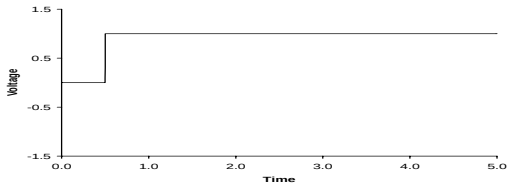


Figure 9 Rising step for lag circuit.

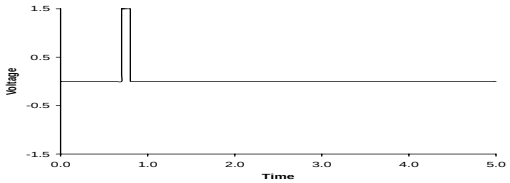
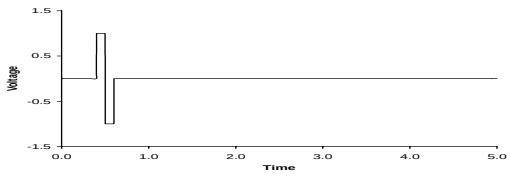


Figure 10 Pulse for lag circuit.



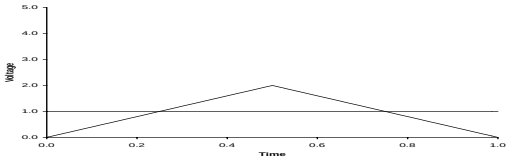


Figure 11 Double pulse for lag circuit.

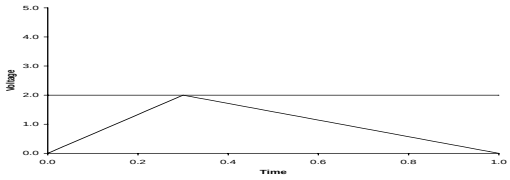


Figure 12 Signal set no. 1.

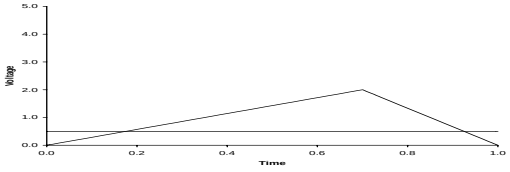


Figure 13 Signal set no. 2.

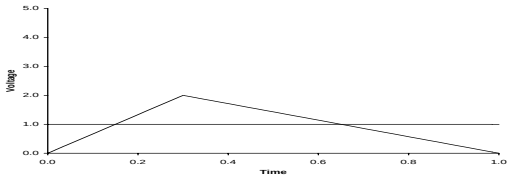


Figure 14 Signal set no. 3.

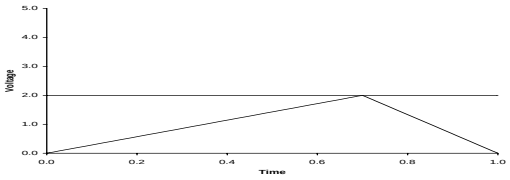


Figure 15 Signal set no. 4.

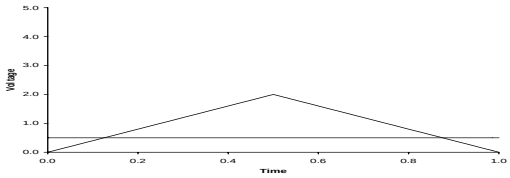


Figure 16 Signal set no. 5.

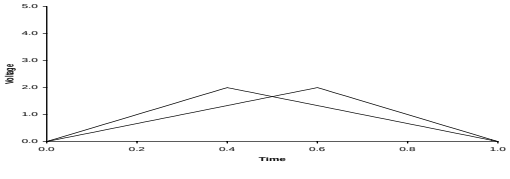


Figure 17 Signal set no. 6.

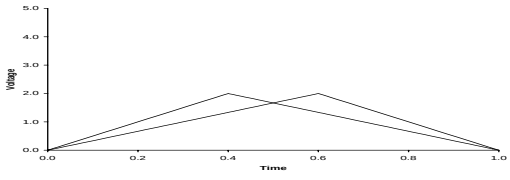


Figure 18 Signal set no. 7.

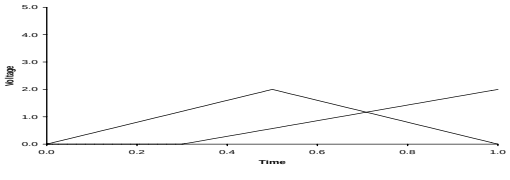


Figure 19 Signal set no. 8.

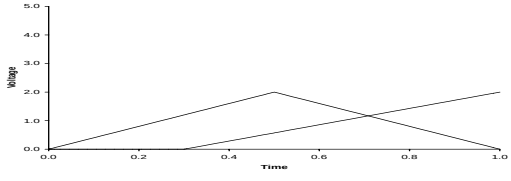


Figure 20 Signal set no. 9.

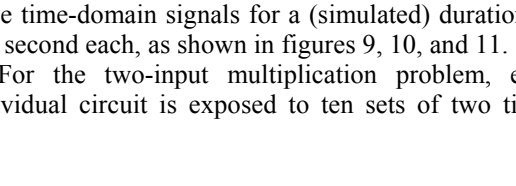


Figure 21 Signal set no. 10.

For the lag problem, the time constant, τ, is held constant at 1.0. Each individual circuit is exposed to three time-domain signals for a (simulated) duration of one second each, as shown in figures 9, 10, and 11.

For the two-input multiplication problem, each individual circuit is exposed to ten sets of two time-domain signals for a (simulated) duration of one second each, as shown in figures 12 through 21. Figure pairs 18 and 19 and figure pairs 20 and 21 show signals where the two inputs are interchanged.

3.6 Control Parameters

For each of the four illustrative problems, the population size, *M*, was 10,000,000. A (generous) maximum size of 500 points (i.e., total number of functions and terminals) was established for the result-producing branch. The percentages of the genetic operations are 60% 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% perturbation on numerical constant terminals, and 9% reproduction. The other parameters are the same default values that we have used previously on a broad range of problems (Koza, Bennett, Andre, Keane 1999).

3.7 Termination

Each run was manually terminated when the values of fitness for successive best-of-generation individuals appeared to have reached a plateau.

3.8 Parallel Implementation

After code development using a single Pentium computer, each of the four illustrative problems 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).

4 Results for Squaring Circuit

The fitness of the best individual from generation 0 is 1195.56 for the problem of designing a squaring circuit.

The best-of-run individual appeared in generation 109. The circuit-constructing program tree has 283 points. This best-of-run circuit has 37 components and a fitness of 5.83147 (figure 22).

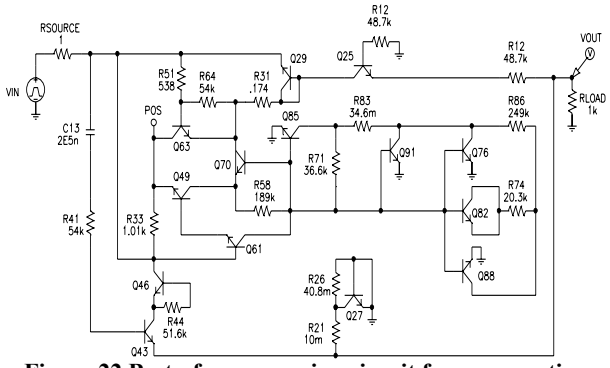


Figure 22 Best-of-run squaring circuit from generation 109.

Figure 23 shows the output voltage produced by the best-of-run individual from generation 109 for the rising ramp input superimposed on the (virtually indistinguishable) correct output voltage for the squaring function.

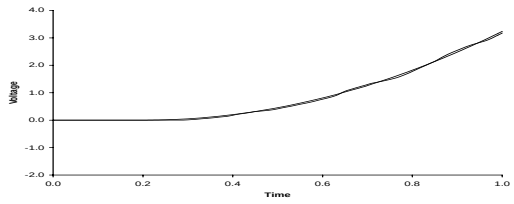


Figure 23 Output for rising ramp input for squaring circuit.

Figure 24 shows the output for the falling ramp input superimposed on the (virtually indistinguishable) correct output voltage for the squaring function.

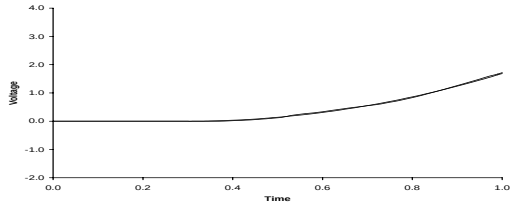


Figure 24 Output for falling ramp input for squaring circuit.

Figure 25 shows the output for the rising step input superimposed on the (virtually indistinguishable) correct output voltage for the squaring function.

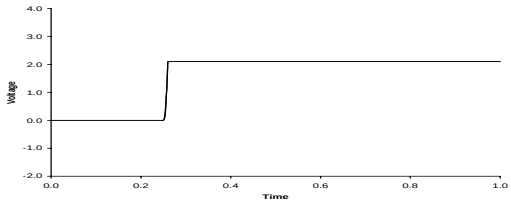


Figure 25 Output for rising step input for squaring circuit.

Figure 26 shows the output for the falling step input superimposed on the (virtually indistinguishable) correct output voltage for the squaring function.

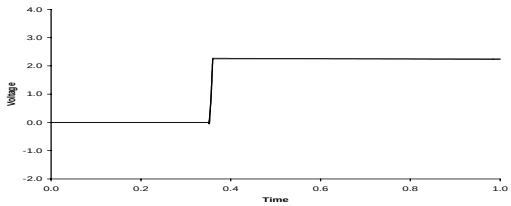


Figure 26 Output for falling step input for squaring circuit.

As can be seen, the output voltage produced by the best-of-run individual from generation 109 closely matches the desired output for a squaring circuit. SPICE employed 293 points in simulating this circuit for the four input signals. The average error over these 293 points was 0.027 volts. All analog computational circuits are, of course, approximate. This small average error would be acceptable for many applications.

4.1 Computer Time for Squaring Circuit

The best-of-run individual from generation 109 for the squaring computational circuit was produced after evaluating 1.1×10^9 individuals. This required 43 hours (1.548×10^5 seconds) on our 1,000-node parallel computer system — that is, the expenditure of 5.418×10^{16} computer cycles (about 54 peta-cycles of computer time).

5 Results for Square Root Circuit

The fitness of the best individual from generation 0 is 212.0 for the problem of designing a square root circuit.

The best-of-run individual appeared in generation 66. The circuit-constructing program tree has 284 points. This best-of-run circuit has 39 components and a fitness of 6.26989 (figure 27).

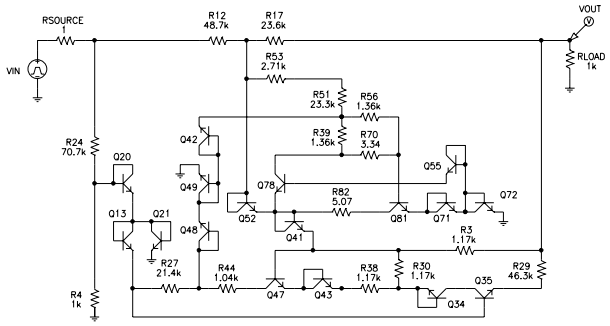


Figure 27 Best-of-run square root circuit from generation 66.

Figure 28 shows the output voltage produced by the best-of-run individual from generation 66 for the rising ramp input superimposed on the (virtually indistinguishable) correct output voltage for the square root function.

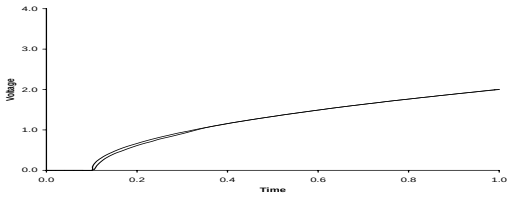


Figure 28 Output for rising ramp input for square root circuit.

Figure 29 shows the output for the triangle input superimposed on the (virtually indistinguishable) correct output voltage for the square root function.

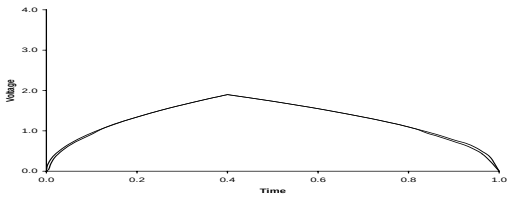


Figure 29 Output for triangle input for square root circuit.

Figure 30 shows the output for the rising step input superimposed on the (virtually indistinguishable) correct output voltage for the square root function.

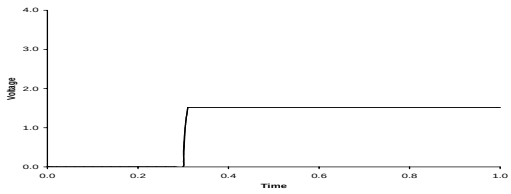


Figure 30 Output for rising step input for square root circuit.

Figure 31 shows the output for the pulse input superimposed on the (virtually indistinguishable) correct output voltage for the square root function.

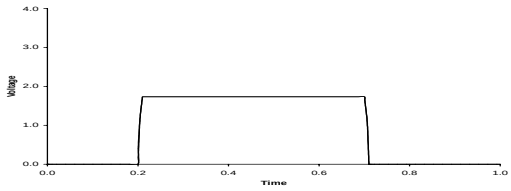


Figure 31 Output for pulse input for square root circuit.

As can be seen from the figures, the output voltage produced by the best-of-run individual from generation 66 closely matches the desired output for a square root circuit. SPICE employed 299 points in simulating this circuit for the four input signals. The average error over these 299 points is 0.020 volts.

5.1 Computer Time for Square Root

The best-of-run individual from generation 66 for the square root circuit was produced after evaluating 6.7×10^9 individuals. This required 52 hours (1.872×10^5 seconds) on our 1,000-node parallel computer system — that is, the expenditure of 6.552×10^{16} computer cycles (about 65 peta-cycles of computer time).

6 Results for Lag Circuit

The fitness of the best individual from generation 0 is 294.71 for the problem of designing a lag circuit.

The best-of-run circuit appeared in generation 71 with a fitness of 1.629 and 29 components.

In this run, for example, the best-of-generation individual from generation 69 has fewer components (26) than the best-of-run individual from generation 71, while having an almost identical value of fitness (1.634). Thus, we designate the best-of-generation from generation 69 (shown in figure 32) as the result of this run. The circuit-constructing program tree for the best-of-generation individual from generation 69 has 99, 141, and 2 points, respectively.

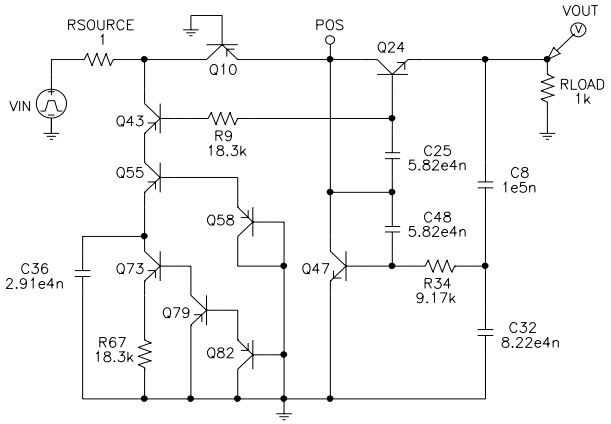


Figure 32 Best-of-generation lag circuit from generation 69.

Figure 33 shows the output voltage produced by the best-of-generation individual from generation 69 for the rising step input superimposed on the (virtually indistinguishable) correct output voltage.

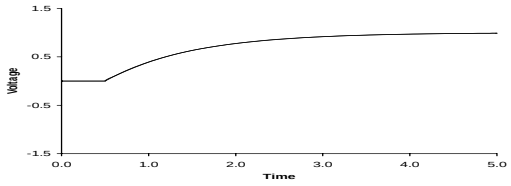


Figure 33 Output for rising step input for lag circuit.

Figure 34 shows the output for the pulse input superimposed on the (virtually indistinguishable) correct output voltage for the lag function.

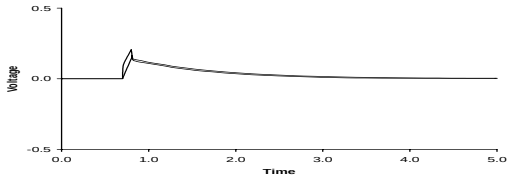


Figure 34 Output for pulse input for lag circuit.

Figure 35 shows the output for the double pulse input superimposed on the (virtually indistinguishable) correct output voltage for the lag function.

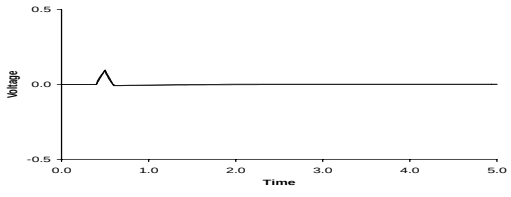


Figure 35 Output for double pulse input for lag circuit.

As can be seen from the figures, the output voltage produced by the best-of-run individual from generation 69 closely matches the desired output for a lag circuit. SPICE employed 202 points in simulating this circuit for the three input signals. The average error over these 202 points was 0.002 volts.

6.1 Computer Time for Lag Circuit

The best-of-run individual from generation 69 for the lag circuit was produced after evaluating 70×10^8 individuals. This required 47 hours (1.692×10^5 seconds) on our 1,000-node parallel computer system — that is, the expenditure of 5.922×10^{16} computer cycles (about 59 peta-cycles of computer time).

7 Results for Multiplier Circuit

The fitness of the best individual from generation 0 is 2,229.2 for the problem of designing a multiplier circuit.

The best-of-run individual appeared in generation 79 with a fitness of 139.07. The three result-producing branches of the circuit-constructing program tree has 277, 17, and 34 points, respectively. This best-of-run circuit has 40 components (figure 36).

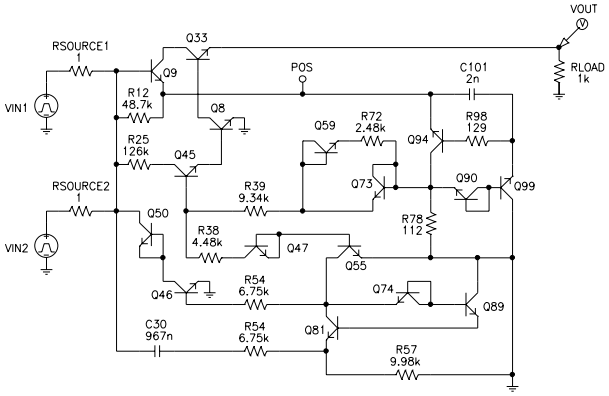


Figure 36 Best-of-run multiplier circuit from generation 79.

Figures 37 through 46 show the output voltage produced by the best-of-run individual from generation 79 for each of the 10 signal sets (figures 12 through 21) superimposed on the correct output voltage.

As can be seen from the 10 figures, the output voltage produced by the best-of-run individual from generation 79 closely matches the desired output for a multiplier circuit. SPICE employed 635 points in simulating this circuit for the ten input signals. The average error over these 635 points was 0.121 volts. The most significant error occurs at the right ends of figures 45 and 46.

7.1 Computer Time for Multiplier Circuit

The best-of-run individual from generation 79 for the multiplier computational circuit was produced after evaluating 8.0×10^9 individuals. This required 141 hours (5.076×10^5 seconds) on our 1,000-node parallel computer system — that is, the expenditure of 1.777×10^{17} computer cycles (about 178 peta-cycles of computer time).

8 Conclusions

This paper used multiple computationally intensive time-domain simulations to automatically synthesize both the topology and sizing for a squaring, square root, and multiplier computational circuit and a lag circuit.

Editor's Note

This paper is a combination and consolidation of originally separate papers.

References

Babanezhad, J. N. and Temes, G. C. 1986. Analog MOS Computational Circuits. *Proceedings of the IEEE Circuits and System International Symposium*. Piscataway, NJ: IEEE Press. Pages 1156–1160.

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., Keane, Martin A., Yu, Jessen, Mydlowec, William, and Stiffelman, Oscar. 1999. Evolution by means of genetic programming of analog circuits that perform digital functions. 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. Pages 1477 - 1483.

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.

Gilbert, Barrie. 1968. A precise four-quadrant multiplier with subnanosecond response. *IEEE Journal of Solid-State Circuits*. Volume SC-3. Number 4. December 1968. Pages 365–373.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. 2nd. Ed. Cambridge, MA: MIT Press.

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

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

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

Koza, John R., 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.

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

Poli, Riccardo, Banzhaf, Wolfgang, Langdon, William B., Miller, Julian, Nordin, Peter, and Fogarty, Terence C. 2000. *Genetic Programming: European Conference, EuroGP 2000, Edinburgh, Scotland, UK, April 2000, Proceedings*. Lecture Notes in Computer Science. Volume 1802. Berlin, Germany: 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.

Sheingold, Daniel H. (editor). 1976. *Nonlinear Circuits Handbook*. Norwood, MA: Analog Devices, Inc.

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.

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

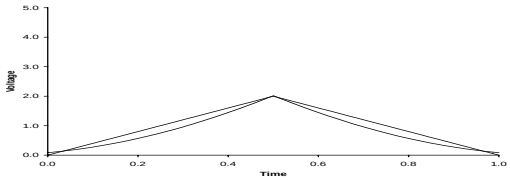


Figure 37 Output for signal no. 1 for multiplier circuit.

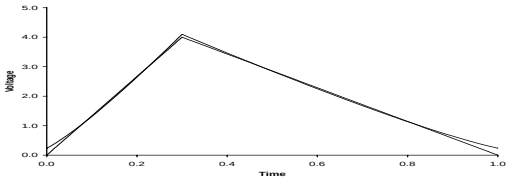


Figure 38 Output for signal no. 2 for multiplier circuit.

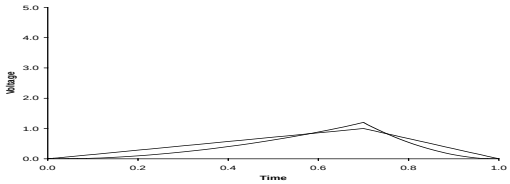


Figure 39 Output for signal no. 3 for multiplier circuit.

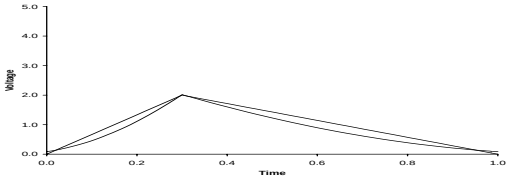


Figure 40 Output for signal no. 4 for multiplier circuit.

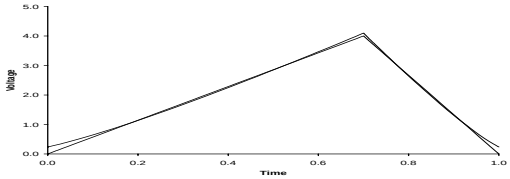


Figure 41 Output for signal no. 5 for multiplier circuit.

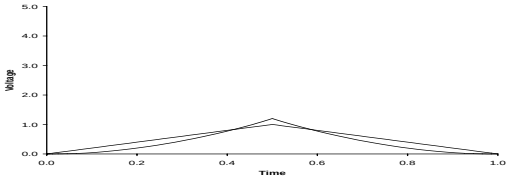


Figure 42 Output for signal no. 6 for multiplier circuit.

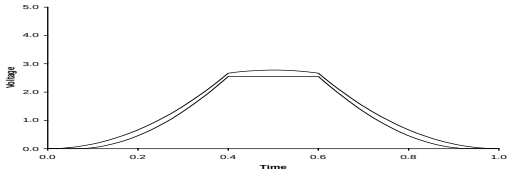


Figure 43 Output for signal no. 7 for multiplier circuit.

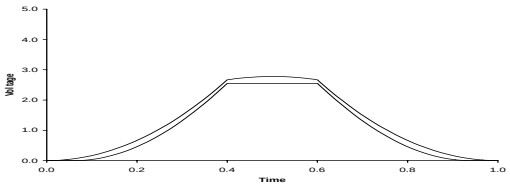


Figure 44 Output for signal no. 8 for multiplier circuit.

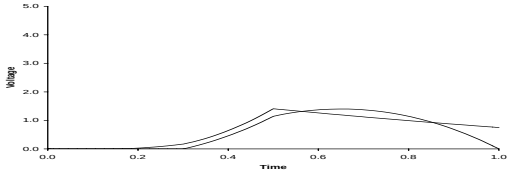


Figure 45 Output for signal no. 9 for multiplier circuit.

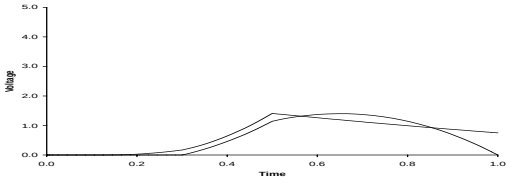


Figure 46 Output for signal no. 10 for multiplier circuit.