Iterative Refinement of Computational Circuits using Genetic Programming

Matthew J. Streeter
Genetic Programming, Inc.
Mountain View, California
mjs@tmolp.com

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

John R. Koza
Stanford University
Stanford, California
koza@stanford.edu

Abstract

Previous work has shown that genetic programming is capable of creating analog electrical circuits whose output equals common mathematical functions, merely by specifying the desired mathematical function that is to be produced. This paper extends this work by generating computational circuits whose output is an approximation to the *error function* associated with an existing computational circuit (created by means of genetic programming or some other method). The output of the evolved circuit can then be added to the output of the existing circuit to produce a circuit that computes the desired function with greater accuracy. This process can be performed iteratively. We present a set of results showing the effectiveness of this approach over multiple iterations for generating squaring, square root, and cubing computational circuits. We also perform iterative refinement on a recently patented cubic signal generator circuit, obtaining a refined circuit that is 7.2 times more accurate than the original patented circuit. The iterative refinement process described herein can be viewed as a method for using previous knowledge (i.e. the existing circuit) to obtain an improved result.

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 total 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, capacitors). 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; Mydlowec and Koza 2000).

This paper extends previous work on synthesis of computational circuits using genetic programming by applying genetic programming in an iterative manner to obtain successively more accurate computational circuits. Specifically, we employ a multiple-run process in which each run involves generating a computational circuit that

produces as output an approximation to the *error function* (i.e. the difference between the target function and the circuit's actual output) of the best-of-run circuit from the previous run.

Section 2 of this paper presents a proof-of-principle experiment involving iterative refinement of rational polynomial approximations to the sine function. Section 3 describes automatic synthesis of electrical circuits by means of developmental genetic programming. Section 4 itemizes the preparatory steps required to apply genetic programming to the problem of synthesizing computational circuits. Section 5 describes our experiments involving iterative refinement of squaring, square root, and cubing computational circuits. Section 6 describes an experiment involving iterative refinement of a post-2000 patented cubic signal generator circuit. Section 7 is the conclusion.

2 ITERATIVE REFINEMENT OF NUMERICAL APPROXIMATIONS TO FUNCTIONS

Iterative refinement as described in this paper can be applied to other problem areas involving the synthesis of structures or entities that generate a specified target curve as output. In particular, it can be applied to the problem of generating numerical approximations to mathematical functions. As a proof-of-principle experiment that could be conducted using relatively modest computational resources, we evolved successively more accurate rational polynomial approximations to the function sin(x) over the interval [0,π/2].

The function set for this problem consisted of the four arithmetic operators {+,*,-,%}. The terminal set consisted of the variable X and the random numeric terminal R. Fitness was defined as the sum of absolute error over 100 uniformly spaced points. We used a population size, *M*, of 100,000 and tournament selection with a tournament size of 10. The remaining control parameters are the same as those described in section 4.6 of this paper. Each run was conducted on a single Pentium workstation.

A first run (iteration 0) was conducted using the target function sin(*x*). The best-of-run individual from this first run produces an output $a_0(x)$, with an associated error function defined as $\sin(x)-a_0(x)$. We then perform a second run (iteration 1) using the target function $\sin(x)-a_0(x)$. This second run yields a best-of-run individual whose output $a_1(x)$ can be added to the result of the first run to produce a more accurate approximation $a_0(x)+a_1(x)$. This process was continued iteratively for a total of 6 runs including the initial run (with target function sin(*x*)). The best-of-run rational polynomial from each of the runs was imported into the Maple symbolic mathematics package to calculate average error (using the integral of the error functions). Table 1 presents the average error of the best-of-run individuals for iteration *N*=0 through iteration *N*=5. R_{prev} indicates the ratio of improvement in average error over the previous iteration. R_{final} at the bottom of the table gives the ratio of improvement in accuracy of the best individual from all iterations over the best individual from the first iteration, i.e. the total ratio of improvement over all iterations.

Table 1 : Error of Rational Polynomial Approximations to sin(x), [0,π/2] over Successive Iterations

N	ERROR	R_{prev}
0	4.554574e-6	—
1	3.483825e-8	130.73
2	2.093228e-8	1.6643
3	1.039170e-8	2.0143
4	6.302312e-9	1.6489
5	2.200276e-6	3.4912e-4

R_{final}: 772.65

As shown in Table 1, we obtain a significantly more accurate approximation for iterations 1 through 4 of our experiment. With each successive iteration, the error function being approximated becomes more complex (i.e. has more local extrema and sharper spikes). On iteration 5, we obtain a high-degree rational polynomial that, as evaluated under our genetic programming system over 100 fixed points, represents an improvement over the previous iteration. However, when imported into a symbolic mathematics package this approximation is revealed to have a large spike in between two of these 100 fixed points, giving it a substantially higher average error than the approximation produced by the previous iteration.

3 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. We use the methods described by Koza, Bennett, Andre and Keane (1999) in combination with the four technical modifications described by Streeter, Keane, and Koza (2002).

4 PREPARATORY STEPS

4.1 INITIAL CIRCUIT

A one-input, one-output initial circuit with one modifiable wire was used for all problems described in this paper with the exception of the problem involving the patented cubic signal generator circuit (described in section 6). The initial circuit has an incoming signal source, a 1 μOhm source resistor, a voltage probe point VOUT, and a 1 gigaOhm load resistor. The topology of the circuit is the same as that shown in figure 30.1 of Koza, Bennett, Andre, and Keane 1999.

4.2 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 each of the problems described in this paper. Automatically defined functions were not used.

4.3 TERMINAL SET

The value of each component in a circuit possessing a parameter (e.g., resistors) is established by an argument of its component-creating function. The argument is a numerical terminal whose value can be perturbed by a special genetic operation for mutation of constants. Aside from the numerical constants, the terminal set for each result-producing branch is

T_{rpb} = {END, SAFE_CUT, B_C_E ... E_C_B, BIFURCATE_POSITIVE, BIFURCATE_NEGATIVE, Q2N3906, Q2N3904, UP_OR_LEFT, DOWN_OR_RIGHT}.

These terminals are each described in detail in Koza, Bennett, Andre, and Keane 1999 and Streeter, Keane, and Koza 2002. Briefly, the END terminal is a development-controlling 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. The B_C_E through E_C_B terminals specify which of six possible permutations to use for the three leads of a transistor when inserting a transistor into a circuit. The BIFUCATE_POSITIVE and BIFURCATE_NEGATIVE terminals specify which end of the modifiable wire to bifurcate when inserting a transistor. The Q2N3906 and Q2N3904 terminals specify which of two available transistor models to use when inserting a transistor. The DOWN_OR_RIGHT and UP_OR_LEFT terminals specify the two possible pairs of directions in which parallel division can be performed.

4.4 FUNCTION SET

The function set for each result-producing branch is

F_{rpb} = {R, Q, SERIES, PARALLEL, NODE, TWO-LEAD, TWO_GROUND, THREE_GROUND}.

See Koza, Bennett, Andre, and Keane 1999 and Streeter, Keane, and Koza 2002 for a detailed description of each of these functions. Briefly, the R function is a component-creating function that creates a resistor with a resistance specified as an argument to the function. A function that creates a two-leaded component (with R here being the only choice) can be used as an argument to the TWO-LEAD function, which inserts two-leaded components into the circuit. The Q function inserts transistors 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 NODE function is used to connect distant points in a circuit. The TWO_GROUND and THREE_GROUND functions each create a via to ground.

4.5 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 intermediate circuits during the developmental process), thereby eventually yielding a fully developed circuit. A netlist is then 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 two time-domain signals for a (simulated) duration of either one or one hundred seconds. The first time domain signal is a straight line rising from 0 to 1 volt over a period of one second. The second time domain signal is a straight line falling from 1 volt to 0 volts over 100 seconds. Each time-domain simulation is run for 100 time steps.

Absolute error is defined as the sum, over each time step, of the absolute difference between the output of the circuit and the value of the target function (which varies from problem to problem). Overall fitness consists of a

weighted sum of absolute error. The error of each point is weighted by a factor of 10 if it is not a hit, and a factor of 1 otherwise. A hit is defined as an output that is within 1% of the desired value.

4.6 CONTROL PARAMETERS

For the computational circuit problems described in this paper, the population size, *M*, was 20,000. 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).

4.7 TERMINATION

Each run (performed as part of a multi-run process) was allowed to perform 100 generations before being terminated.

4.8 PARALLEL IMPLEMENTATION

Each of the problems described in this paper 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 20 350 MHz Pentium II processors (each accompanied by 64 megabytes of RAM). These 20 processors were isolated into a logically separate cluster for the purpose of these experiments. They normally act as part of a larger 1,000 processor cluster which we apply to more difficult problems involving genetic programming. 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 = 1,000 at each of D = 20 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. Emigrants are selected randomly and the migration rate is 5% (10% if the adjacent node is in the same physical box).

5 ITERATIVE REFINEMENT OF EVOLVED COMPUTATIONAL CIRCUITS

5.1 RESULTS

We applied our iterative refinement process to the generation of squaring, square root, and cubing computational circuits. In each case, a first run of genetic programming was conducted to create a circuit which output the desired function with some degree of accuracy. A second run was then performed in which the target function was taken as the error function associated with the best-of-run circuit from the previous run. This process was continued for successive iterations until a run produced an improvement in accuracy over the previous iteration of 5% or less. Tables 2, 3, and 4 present the results of these experiments for squaring, square root, and cubing computational circuits, respectively. Figures 1, 3, and 5 present the output curves for the circuits produced by successive iterations for the squaring, square root, and cubing functions, respectively. Figures 2, 4, and 6 present the error curves over successive iterations for the squaring, square root, and cubing circuits, respectively.

Table 2 : Error of Squaring Computational Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	46.84	—
1	5.836	8.0260
2	5.5635	1.0490

R_{final}: 8.4193

Table 3 : Error of Square Root Computational Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	11.835	—
1	3.4445	3.4359
2	3.398	1.0137

R_{final}: 3.4830

Table 4 : Error of Cubing Computational Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	22.312	—
1	20.198	1.1047
2	17.510	1.1535
3	17.061	1.0263

R_{final}: 1.3078

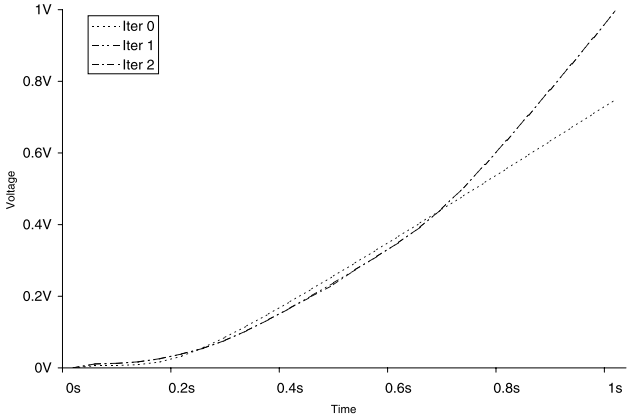

Figure 1: Output of Squaring Computational Circuits over Successive Iterations

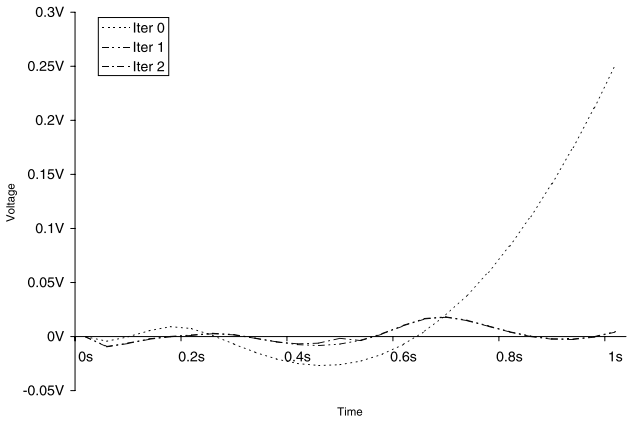

Figure 2: Error of Squaring Computational Circuits over Successive Iterations

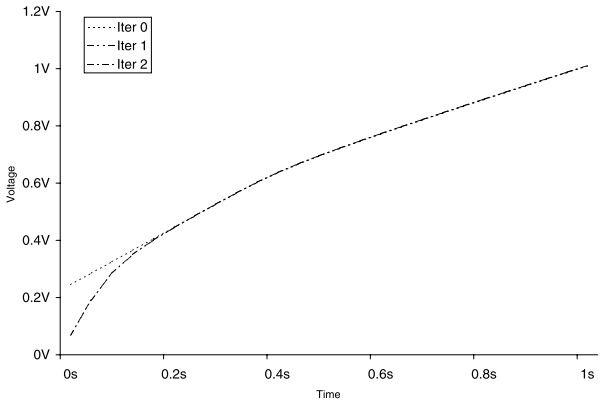

Figure 3: Output of Square Root Computational Circuits over Successive Iterations

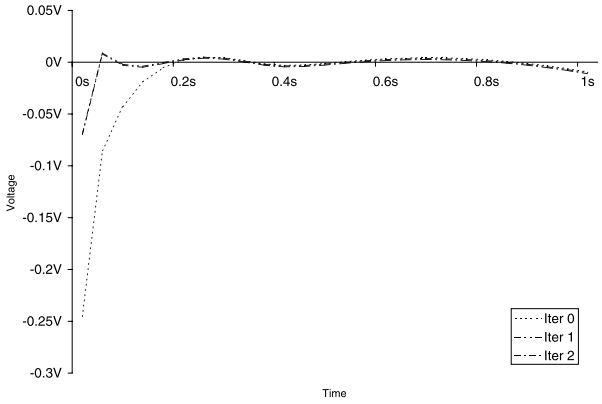

Figure 4: Error of Square Root Computational Circuits over Successive Iterations

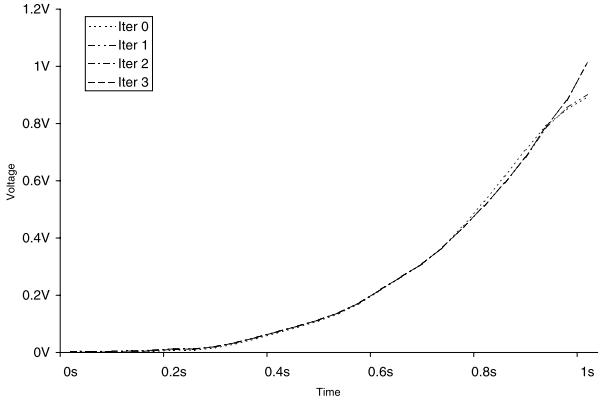

Figure 5: Output of Cubing Computational Circuits over Successive Iterations

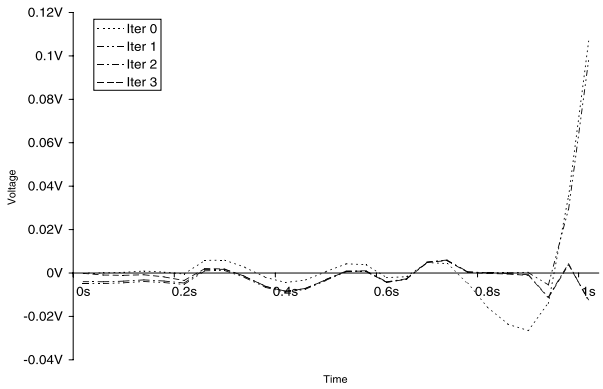

Figure 6: Error of Cubing Computational Circuits over Successive Iterations

5.2 EFFICIENCY OF EVOLVING COMPUTATIONAL CIRCUITS USING ITERATIVE REFINEMENT

It is not necessarily (and almost undoubtedly not) the case that the experiments described above produce their results with the most efficient possible use of available computational resources. Since our multi-run process involved 6 separate GP runs, the same computational resources could be expended by executing (for example) a single run for 6 times as long or a single run using 6 times the population. We have no evidence to show that either of these approaches, or some other approach which used the same amount of computational resources as our experiments, would not produce better results than the ones given here. However, our purpose here is to establish that genetic programming is capable of creating computational circuits whose output is an approximation to the error function of another computational circuit, and that the iterative process used in these experiments is indeed able to create successively more accurate computational circuits.

Also note that the results presented in this section and in section 2 in no way imply that the error functions being targeted in each successive iteration are somehow more amenable to approximation by genetic programming than are the original target functions (i.e. sqrt(x), x^2, and x^3). Rather, it is the fact that genetic programming is able to produce circuits whose outputs approximate the error functions with any degree of accuracy at all that allows successively more accurate circuits to be obtained.

6 REFINEMENT OF A POST-2000 PATENTED CIRCUIT

U.S. patent 6,160,427 covers a low-voltage cubic signal generator circuit that produces an output current approximately equal to the cube of its input current (Cipriani and Takeshian, 2000). In a previous paper, we successfully applied genetic programming to the synthesis of a computational circuit that duplicates the functionality of this patented circuit. Specifically, we evolved a circuit that meets the low voltage criteria defined in the patent and has an average error that is 59% of that of the patented circuit over the four fitness cases on which it was evaluated (Streeter, Keane, and Koza 2002).

We now apply our iterative refinement process to this patented circuit. The preparatory steps for the runs performed in this experiment differ from the preparatory steps given in section 4 in four minor ways. First, we force the evolved circuit to conform to the low voltage specification in the patent by including only a 2V power supply in the function set, rather than the 15V and –15V supplies we had included in previous run. Second, we desire that this circuit make full use of the voltage range provided by the power supply, i.e. the circuit's output should range from 0V to 2V. This means that the input voltage should have a minimum value of 0V and a maximum voltage equal to the cube root of 2. The two time-domain curves described in section 4.5 are modified accordingly. Third, for this problem we use only a minimal embryo consisting of a single modifiable wire initially not connected to the test fixture, in contrast to the embryo consisting of a modifiable wire connected to source and load resistors as described in section 4.1. We generally use this minimal embryo when dealing with a test fixture that has multiple inputs or outputs. In this case there is one input (representing an input current flowing down from a 2V supply) and two outputs (representing the high and low points at which the output current is to be probed). Additional functions are included in the function set that allow connections to be made to the single input and the two outputs. For more information on these functions see Koza, Bennett, Andre, Keane 1999. Finally, each of the runs described in this section was manually monitored and terminated when it appeared to have reached a plateau.

When building upon a patented circuit, the first iteration (iteration 0) of our iterative process does not involve a GP run, but rather involves building and simulating the patented circuit based on the schematics given in the patent document. Our first GP run (iteration 1) involves the synthesis of a computational circuit whose output approximates the error function associated with the patented circuit. Our second GP run (iteration 2) involves the synthesis of a circuit whose output approximates the error function of the best-of-run circuit from our first GP run. The results of these runs are presented in Table 5.

Table 5 : Error of Low-Voltage Cubing Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	7.128	—
1	0.9873	7.2197
2	0.9236	1.0690

R_{final}: 7.2197 (see below)

The first iteration of our experiment produced a circuit whose average error was 7.2 times better than that of the patented circuit (over our two fitness cases). This circuit was created after 147 generations. The circuit was tested on a variety of unseen fitness cases, and outperformed the patented circuit on those fitness cases as well. The second iteration of our experiment produced a circuit that was approximately 1.07 times better than the circuit of the previous iteration. This circuit was created after 80 generations. However, examination of the output of this circuit reveals that it contains a number of sharp spikes which indicate instability in the circuit's output. This circuit generally does not perform better that the circuit of the previous iteration when tested on unseen fitness cases.

The output curves for the circuits produced in iterations 0 and 1 of our experiment are given in Figure 7. The error curves for these two circuits are given in Figure 8.

Figure 9 presents the refined computational circuit obtained by adding the output of the patented cubic signal generator circuit to the output of the evolved circuit. In this figure there are two rectangles drawn using dotted lines. The top rectangle encloses the circuitry from the patented cubic signal generator, while the bottom rectangle encloses the evolved circuitry. Components which are outside both rectangles are part of the test fixture for the problem. The two outputs labeled `VITER0` (for the output of the patented circuit) and `VITER1` (for the output of the evolved circuit) are passed through a voltage addition block to produce the final output (`VFINAL`) of the refined circuit.

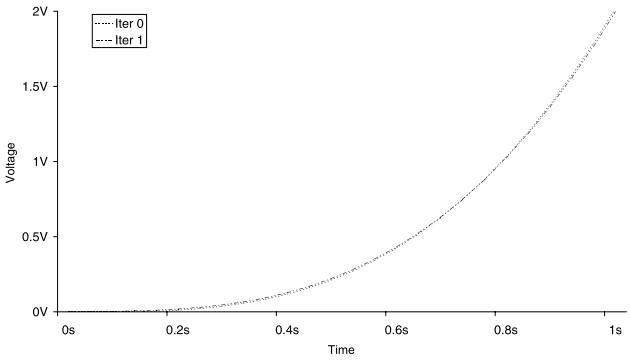

Figure 7: Output of Low-Voltage Cubing Circuits over Successive Iterations

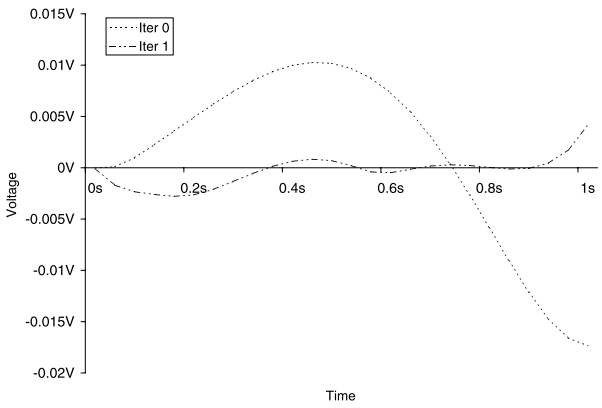

Figure 8: Error of Low-Voltage Cubing Circuits over Successive Iterations

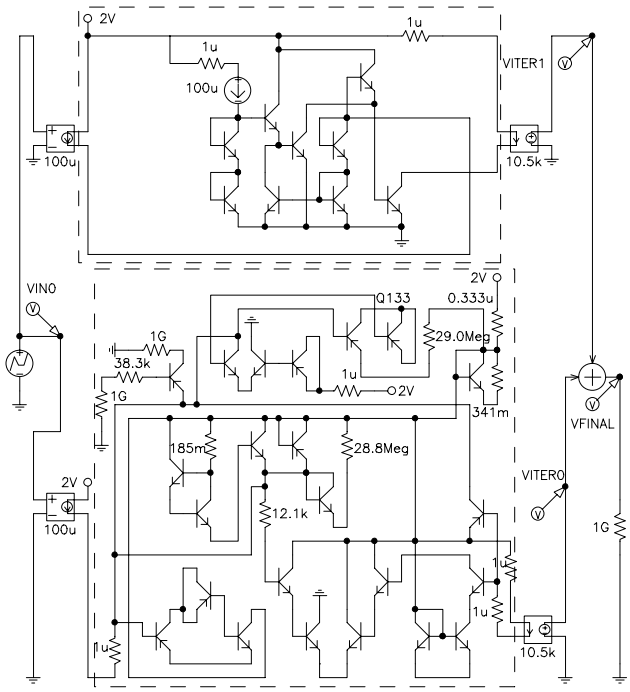

Figure 9: Refined Low-Voltage Cubic Signal Generator Circuit

As previously mentioned, computational circuits are valuable in that they are able to produce the output of complex mathematical functions at near-instantaneous analog speed. However, no human-designed computational circuit (or circuit designed by any other means) is likely to produce the desired output with perfect accuracy. For this reason, an error correction process is desireable. Genetic programming has been shown in this paper to provide such a process. Since the shape of the error functions associated with computational circuits are rather arbitrary and do not correspond to simple mathematical functions, it would be difficult if not impossible for an analog engineer to generate a circuit that produced these functions. The synthesis of computational circuits that produce these error functions as output is therefore an especially suitable application of genetic programming.

7 CONCLUSIONS

We have shown that an iterative refinement process can be applied both to the generation of rational polynomial approximations to functions and to the synthesis of computational circuits using genetic programming. In particular, we have applied this process to the synthesis of squaring, square root, and cubing computational circuits, obtaining an increase in accuracy with each successive iteration of refinement. We further applied this iterative process to a recently patented low-voltage cubic signal generator circuit, and achieved an improvement in accuracy of a factor of 7.2 over the patented circuit. We conclude that the approach described in this paper provides an effective way to generate high-accuracy computational circuits.

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.

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.

Cipriani, Stefano and Takeshian, Anthony A. 2000. *Compact cubic function generator*. U. S. patent 6,160,427. Filed September 4, 1998. Issued December 12, 2000.

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.

Mydlowec, William and Koza, John R. 2000. Use of Time-Domain Simulations in Automatic Synthesis of Computational Circuits using Genetic Programming. In Whitley, Darrell (editor). 2000. *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*. Las Vegas, NV: Morgan Kaufmann. 187-197.

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.

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

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.

Streeter, Matthew J., Keane, Martin A., and Koza, John R. 2002. Routine Duplication of Post-2000 Patented Inventions by Means of Genetic Programming. In Lutton, Evelyn, Foster, James A., Miller, Julian, Ryan, Conor and Tettamanzi, Andrea. *Proceedings of EuroGP'2002, April 3-5, 2002, Kinsale, Ireland*. Springer-Verlag. 26-36.