Self Modifying Cartesian Genetic Programming: Finding Algorithms that Calculate pi and e to Arbitrary Precision

Simon Harding
Department Of Computer Science
Memorial University
Newfoundland, Canada
simonh@cs.mun.ca

Julian F. Miller
Department of Electronics
University of York
York, UK
jfm7@ohm.york.ac.uk

Wolfgang Banzhaf
Department Of Computer Science
Memorial University
Newfoundland, Canada
banzhaf@cs.mun.ca

ABSTRACT

Self Modifying Cartesian Genetic Programming (SMCGP) aims to be a general purpose form of developmental genetic programming. The evolved programs are iterated thus allowing an infinite sequence of phenotypes (programs) to be obtained from a single evolved genotype. In previous work this approach has already shown that it is possible to obtain mathematically provable general solutions to certain problems. We extend this class in this paper by showing how SMCGP can be used to find algorithms that converge to mathematical constants (pi and e). Mathematical proofs are given that show that some evolved formulae converge to pi and e in the limit as the number of iterations increase.

Categories and Subject Descriptors

I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming; D.1.2 [**Software**]: Automatic Programming

General Terms

Algorithms

Keywords

Genetic programming, developmental systems

1. INTRODUCTION

Self Modifying Cartesian Genetic Programming (SMCGP) is a form of developmental genetic programming, based on Cartesian Genetic Programming [9]. The concept is that CGP programs, which are directed graphs, contain not only functions for computation, but also functions that can change the program during run time [3].

SMCGP has previously been applied to a number of different tasks including finding scalable, general solutions to digital circuits [6], finding sequences and mathematical results [5] and evolving learning algorithms [4].

Here, we demonstrate the use of SMCGP to find programs that approximate the fundamental constants π (3.1415...) and e (2.7182...). Two different approaches are used, one is to find a program that acts as a mathematical approximation, the other is to find a program that outputs the digits as a sequence.

We provide proofs for two of the evolved formulae (one for pi and one for e) that they rapidly converge to the constants in the limit of large iterations. We consider this work to be significant as evolving provable mathematical results is a rarity in evolutionary computation. Streeter and Becker used tree-based GP to discover mathematical approximations to well known mathematical series, such as the Harmonic series and also new *Padé* approximants to mathematical functions [13]. However, they do not find exact analytical results that can be shown to converge in the limit[1]. Schmidt and Lipson used GP to discover the known Hamiltionians and Langrangians of mechanical systems purely by using GP symbolic regression techniques on data acquired through motion tracking [11]. Schmidt and Lipson investigated using GP for solving iterated function problems (i.e. $f(f(x)) = g(x)$) and they show that one evolved function provably makes $f(f(x))$ converge to $x^2 - 2$ in the limit [10]. Spector et al showed that GP could be used to evolve hitherto unknown algebraic expressions that are important in the mathematics of finite algebras and are orders of magnitude shorter than those that could be produced by prior mathematical methods [12].

The plan of the paper is as follows. In section 2 we discuss the SMCGP technique and the recent improvements made to the previously published method. We discuss and compare results for various ways of applying inputs (terminals) to the SMCGP programs. The SMCGP function set consists of both computational functions and self-modifying functions. These are discussed in section 3. We discuss two distinct methods and results for evolving algorithms that could approximate π in sections 4 and 5. For one method we show that SMCGP can rapidly find potentially novel and fast converging mathematical approximations to π. Our second approach where the digits were evolved as a sequence

[1]Indeed, they note on page 281 that finding such results would be a "striking and exciting application of genetic programming"

is not as effective, but still a plausible methodology for approximating π. In section 6 we discuss experiments and results for the case of approximations to e, and find an algorithmic approximation that is similar to the well known Bernoulli equation. We close with conclusions and future work.

2. SMCGP

As in CGP, in SMCGP each node in the directed graph represents a particular function and is encoded by a number of genes. The first gene encodes the function of the node. This is followed by a number of connection genes (as in CGP) that indicate the location in the graph where the node takes its inputs from. However SMCGP also has three real-valued genes which encode parameters that may be required for the function (primarily self modification (SM) functions use these and in many cases they are truncated to integers when necessary, see later). Section 3 details the available functions and any associated parameters. In this paper all nodes take two inputs, hence each node is specified by seven genes.

As in CGP, nodes take their inputs in a feed-forward manner from either the output of a previous node or from a program input (terminal). We use *relative addressing* in which connection genes specify how many nodes back in the graph they are connected to. Inputs are acquired to programs through the use of special node functions, that we call INP. Outputs are generated using OUTPUT functions.

The evaluation of a genotype is as follows: The initial phenotype is a copy of the genotype. This graph is then executed, and if there are any modifications to be made, they alter the phenotype graph.

The genotype is invariant during the entire evaluation of the individual. When executed, the phenotype is initially a copy of the genotype, and all modifications are made to the phenotype. In subsequent iterations, the phenotype will usually gradually diverge from the genotype. The encoded graph is executed in the same manner as standard CGP, but with changes to allow for self-modification. The graph is executed by recursion, starting from the output nodes down through the functions, to the input nodes. In this way, nodes that are unconnected are not processed and do not affect the behavior of the graph at that stage. For function nodes (e.g. +,-,/,*) the output value is the result of the mathematical operation on input values.

Each active (i.e. expressed) graph manipulation function (starting on the leftmost node of the graph) is added to a "To Do" list of pending modifications. After each iteration, the "To Do" list is parsed, and all manipulations are performed (provided they do not exceed the number of operations specified in the user defined "To Do" list length). The parsing is done in order of the instructions being appended to the list, i.e. first in is first to be executed. The length of the list can be limited as manipulations are relatively computationally expensive to perform. Here we limit the length to just 2 instructions, which simplifies human analysis. All graph manipulation functions use extra genes as parameters. This is described in section 3. Complete details of SMCGP can be found in [4].

We use an (1+4) evolutionary strategy for the experiments in this paper as in CGP. We have used a relatively high (for CGP) mutation rate of 0.1. In these experiments, for simplicity, we chose to make all the rates the same. Mutations for the function type and relative addresses themselves are unbiased; a gene can be mutated to any other valid value.

For the real-valued genes, the mutation operator can choose to randomize the value (with probability 0.1) or add noise (normally distributed, $\sigma 20$). The evolutionary parameters we have used have not been optimized in any way, so we expect to find much better values through empirical investigations.

Evolution is limited to 10,000,000 evaluations. Trials that fail to find a solution in this time are considered to have failed.

3. FUNCTION SET

The function set is defined in two parts. The computational operations as defined in Table 1. The other part is the set of modification operators. These are common to all data types used in SMCGP.

The self-modifying genotype (and phenotype) nodes contain three double precision numbers, called "parameters". In the following discussion we denote these P_0,P_1,P_2. We denote the integer position of the node in the phenotype graph that contained the self modifying function (i.e. the leftmost node is position 0), by x. In the definitions of the SM functions we often need to refer to the values taken by node connection genes (which are all relative addresses). We denote the jth connection gene on node at position i, by c_{ij}.

The modification functions (with the short-hand name) are defined in Table 1.

4. EVOLVING APPROXIMATIONS TO π

There exist several iterative approaches to approximating π.

For example, the Gregory-Leibniz series:

$$\pi = \frac{4}{1} - \frac{4}{3} - \frac{4}{5} - \frac{4}{7} - \frac{4}{9} \ldots$$

This series is simple, but requires a large number of iterations to reach good accuracy. Another method uses recursion to find an approximation:

$$\pi = n(\tan(\pi/n) - \frac{\tan^3(\pi/n)}{3} - \frac{\tan^5(\pi/n)}{5} - \frac{\tan^7(\pi/n)}{7} \ldots)$$

for $n = 1, 2, ...$

Curiously, there has been little work on evolving approximators π, despite it being a well defined problem with many human designed solutions to compare against.

In [7], Krohn used an artificial developmental system based on fractal proteins [1] to produce approximations to π using two different approaches. The first approach was to generate the digits as a binary sequence. The second, and more successful, approach was to use the output of the developmental system to provide values for the equation:

$$\sum_{i=1}^{I} \frac{\sum_{n=2}^{N} B_{n,i}}{\prod_{t=1}^{i} B_{1,i}}$$

where I is the number of developmental iterations, N is the number of behavioural genes (an output of the evolved program) and $B_{n,i}$ is the output of the n^{th} behavioural gene at iteration i.

Basic	
Delete (DEL)	Delete the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$.
Add (ADD)	Add P_1 new random nodes after $(P_0 + x)$.
Move (MOV)	Move the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.
Duplication	
Overwrite (OVR)	Copy the nodes between (P_0+x) and (P_0+x+P_1) to position (P_0+x+P_2), replacing existing nodes in the target position.
Duplication (DUP)	Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.
Duplicate Preserving Connections (DU3)	Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$. When copying, this function modifies the c_{ij} of the copied nodes so that they continue to point to the original nodes.
Duplicate and scale addresses (DU4)	Starting from position (P_0+x) copy (P_1) nodes and insert after the node at position $(P_0 + x + P_1)$. During the copy, c_{ij} of copied nodes are multiplied by P_2.
Copy To Stop (COPYTOSTOP)	Copy from x to the next "COPYTOSTOP" or 'STOP" function node, or the end of the graph. Nodes are inserted at the position the operator stops at.

Connection modification	
Shift Connections (SHIFTCONNECTION)	Starting at node index $(P_0 + x)$, add P_2 to the values of the c_{ij} of next P_1.
Shift Connections 2 (MULTCONNECTION)	Starting at node index (P_0 + x), multiply the c_{ij} of the next P_1 nodes by P_2.
Change Connection (CHC)	Change the $(P_1 mod 3)$th connection of node P_0 to P_2.
Function modification	
Change Function (CHF)	Change the function of node P_0 to the function associated with P_1.
Change Parameter (CHP)	Change the $(P_1 mod 3)$th parameter of node P_0 to P_2.

Numeric functions	
No operation (NOP)	Passes through the first input.
Add, Subtract, Multiply, Divide (DADD, DSUB, DMULT, DDIV)	Performs the relevant mathematical operation on the two inputs.
Const (CONST)	Returns a numeric constant as defined in parameter P_0.
$\sqrt{x}$,$\frac{1}{\sqrt{x}}$ Cos, Sin, TanH, Absolute (SQRT, DRCP, COS, SIN, TANH, DABS)	Performs the relevant operation on the first input (ignoring the second input).
Average (AVG)	Returns the average of the two inputs.
Node index (INDX)	Returns the index of the current node. 0 being the first node.
Input count (INCOUNT)	Returns the number of program inputs.
Min, Max (MIN, MAX)	Returns the minimum/maximum of the two inputs.

Table 1: The function set.

4.1 Fitness Function

Our fitness function was configured to produce a program where subsequent iterations of the program produced more accurate approximation to π. Programs were allowed to iterate for a maximum of 10 iterations. If the output after an iteration did not better approximate π, evaluation was stopped and a large fitness penalty applied. Note that it is possible that after the 10 iterations the output value diverges from π, and the quality of the result would therefore worsen.

The fitness score of an individual is defined as the absolute error of the last output. In addition, the string representation of the output was checked to ensure that all digits matched correctly. Using doubles in .Net limits the precision to 14 decimal places (i.e. 3.14159265358979).

Four variants of the fitness function were tested, each with different configurations of inputs given to the programs, these are:

A One input : the current iteration. Functions can be built using the current iteration counter as a parameter.

B One input : numeric constant (1). The program has no real input, and therefore has to build a structure that performs the iterative process.

C Two inputs : the current iteration and last outputted value. This form can, in some sense, be viewed as recurrent, as programs can depend on the previous output.

D Two inputs : numeric constant (1) and last outputted value. This form can also be viewed as recurrent, as programs can depend on the previous output.

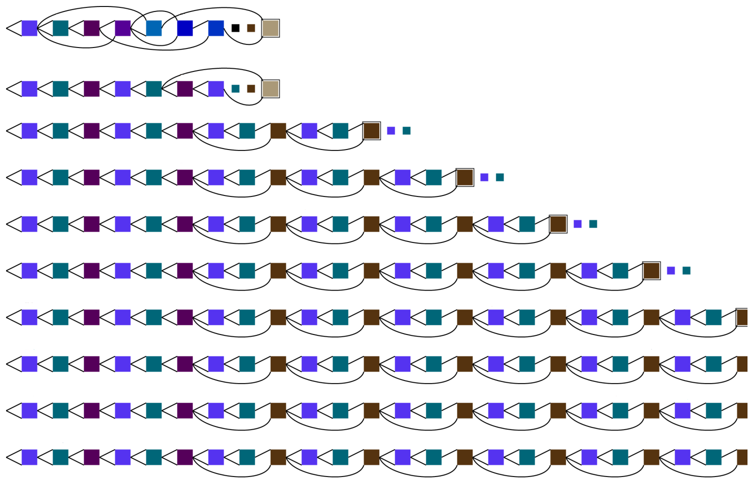

Figure 1: Visualization of SMCGP program that produces an approximation to π. Each row is a different time step.

4.2 Results

The statistical results for these experiments are shown in Table 2. Each experiment was conducted approximately 150 times. The standard deviations are large and overlap, which means that the algorithms appear to perform similarly.

Config.	% Success	Avg. Evals	Min., Avg. Iterations
A	96.7	8,952,441	3, 5.95
B	99.4	2,905,673	3, 5.49
C	96.2	4,953,518	3, 6.78
D	98.7	2,146,348	3, 5.90

Table 2: Finding π using various inputs to the evolved programs. Experiment types: (A)One input, the current iteration; (B)One input, numeric constant (1); (C) Two inputs,the current iteration and last outputted value. (D): Two inputs, numeric constant (1) and the last outputted value. Iterations refers to the number of times the program has to be iterated before it reaches π to 14 decimal places.

4.3 Example π Generator

Figure 1 shows the output of an evolved SMCGP program that accurately converges to π. Table 3 shows the output of the program at each time step. As the program is relatively short, it was possible to extract the evolved generating function:

$$f(i) = \begin{cases} cos(sin(cos(sin(0)))) & i = 0 \\ f(i-1) + sin(f(i-1)) & i > 0 \end{cases} \quad (1)$$

Equation 1 is a nonlinear recurrence relation. However it can be shown that it converges rapidly to π. When $i = 10$, the output matches the first 2048 digits of π.

To prove mathematically that when Eqn 1 is iterated it converges exactly to pi we note that the value of π is a fixed point of equation 1 since $x = x + sinx$ is obeyed when $x = \pi$. It is stable since $f'(x) = 1 + cos(x) = 0$ when $x = \pi$. How rapidly it converges to π can be seen from the following

Iter.	Output Error	Output	Correct
0	3.142	0	0
1	2.475	0.666366745392881	0
2	0.898	2.24379742136643	0
3	0.116	3.02575192260092	0
4	2.589E-4	3.14133374812244	3
5	2.892E-12	3.1415926535869	11
6	0	3.14159265358979	14
7	0	3.14159265358979	14
8	0	3.14159265358979	14
9	0	3.14159265358979	14

Table 3: Output from program shown in figure 1. Output is error (to 3 decimal places) is the difference between π (.Net's Math.PI) and the output from the program. Correct digits is the count of the correctly matching digits after the decimal point. Errors will appear to be 0 when the actual error is very small.

argument. Suppose at some iteration m, $f(m)$ is close to π. Then we can write $f(m) = \pi - \delta$, where δ is a small quantity. Then from Eqn. 1 $f(m+1) = \pi - \delta + sin(\pi - \delta) \approx \pi - \frac{\delta^3}{3!}$. So it approaches pi cubically.

5. EVOLVING THE DIGITS OF π AS A SEQUENCE

Evolving patterns has been a focus of much work in developmental systems (i.e. French flags). Here we take the view that pi could be also be viewed as a one dimensional complex pattern (of repeating digits). Thus we formulate the task here to be that of finding a program that on each iteration will output the next digit of π. The way this is done is as follows. The first time the program is executed it outputs 3. Then the self modification is applied, the program executed again, and it should output 1. Then 4,1,5, and so on.
The program inputs were set to the iteration, i and the previous output value. The function set used is the same as that described above but used with an integer data type.

Iteration	Description
0	CTS (Copy To Stop) returns the constant 3.29, but as the program is interpreted as integers, the value is truncated to 3.
1	INP returns the first input, which is the current iteration (1). The CTS node returns the DSQRT (square root) of 1.
2	CTS returns the truncated value of the constant i.e. 4.
3	The CTS node returns the square root of the square root of the current iteration (3). As a truncated integer, this is 1.
4	Again, the CTS node returns a value (5) from a constant.
5	Here the output comes from adding two constants 4 and 5, to return 9.
6	Here the CTS node connects to a MOV (Move) function which is returning the square root of 6 (as integer) i.e 2.
7	The output (6) is the sum (DADD) of 4 and 2 (which is the integer square root of 7).
8	Here, the output comes from the average (AVG) of 4 and 6 (via the same calculations as iteration 7), to get 5.
9	Here the input value is 9, so the square root function now outputs 3. The left most CTS function returns 3 (via the MOV node connected to the top input). This is because of the order of modification nodes has reached a limit on the ToDo list, and the operation has failed - changing which of the inputs is selected as the output.

Table 4: Description of operations occurring in figure 2.

The fitness function stopped iterating the program when either an incorrect digit or a number < 0 or > 9 was output. Fitness is defined as the total number of correct digits output before making a mistake. Evolution was allowed to continue for 10,000,000 evaluations (or 100 digits of π).

5.1 π Results

The experiment was repeated 310 times (with 31 computers running 10 experiments each). The longest sequence found was 31 correct digits. The shortest was 5 correct digits, and the average number of correct digits was 14.

Figure 2 shows the development stages of the first 10 steps of this program, and a description of each step can be found in table 4. Each step outputs the next digit of π, starting with 3 in the first step. The program consistently uses the final node, a Copy To Stop (CTS) function, as its output. This is because no OUTPUT nodes were used, so the graph runner defaulted to using the last node in the graph as output.

6. EVOLVING APPROXIMATIONS TO E

Aside from π, e is one of the most famous constants in mathematics. There is a well known approximation to e given by (often known as Bernoulli's formula) [8] :

$$\lim_{y\to\infty}(1+\frac{1}{y})^y$$

This series approaches e rather slowly. However recent faster approximations have been found by Brothers and Knox [2]. For instance:

$$\lim_{y\to\infty}(y+1)^{\frac{11y}{6}}(y-1)^{\frac{5y}{6}}(\frac{2y+1}{2y^{y+1}})^{\frac{8}{3}}$$

6.1 Results

Using the same fitness function as with π, evolving solutions for e was found to be significantly harder, the success rate is shown in table 6. However, high quality approximations were found as detailed in the following section. It is suspected that the use of powers in the calculation produces nonlinearity in the fitness scoring, which makes evolution less able to find solutions.

Iteration	Output Error	Output	Digits
0	0.470	2.248310938642	0
1	4.162E-4	2.71786562898987	2
2	2.723E-5	2.71825460295116	4
3	1.722E-6	2.71828010695015	5
4	1.079E-7	2.71828172054962	6
5	6.749E-0	2.71828182170977	8
6	4.219E-10	2.71828182803714	9
7	2.637E-11	2.71828182843268	10
8	1.648E-12	2.7182818284574	11
9	1.030E-13	2.71828182845894	11
10	6.2178E-15	2.71828182845904	13
11	4.441E-16	2.71828182845904	13
12	0	2.71828182845905	14

Table 5: Output from program shown in figure 3. Output is error (to 3 d.p.) is the difference between ei (.Net's Math.E) and the output from the program. Digits is the count of the correctly matching digits after the decimal point.

Config.	Correct digits	Successes	Runs
A	3.75	6	528
B	3.79	2	532
C	3.61	5	542
D	3.59	1	526

Table 6: Finding e using various inputs to the evolved programs. The success rate was very low, with the best performing approach being A with 6 successful solutions found in 528 runs. A successful solution occurs when 14 digits are matched.

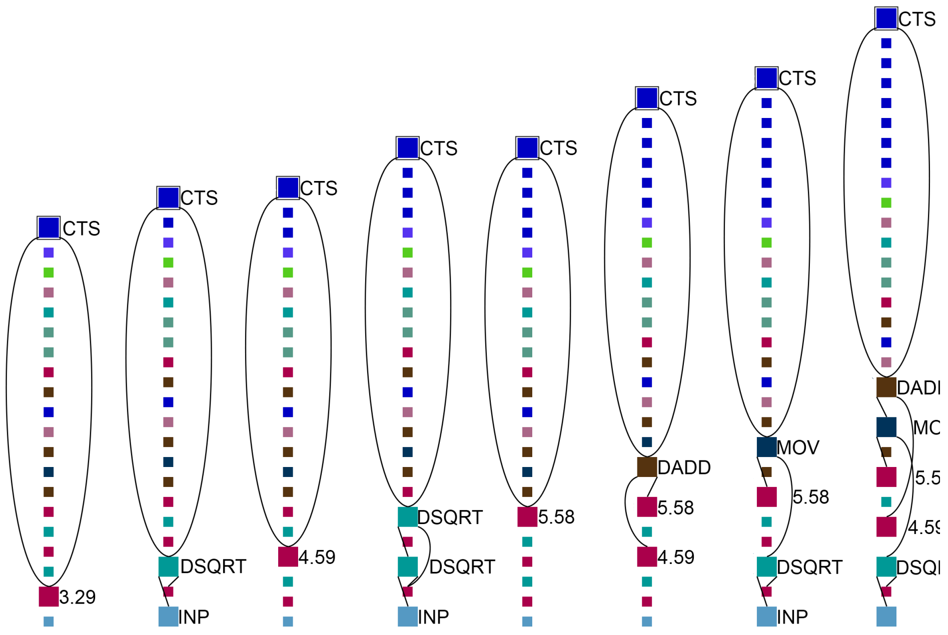

Figure 2: The first 10 developmental steps of a program that produces a π digits sequence.

6.2 Example e Generator

Figure 3 shows the output of an evolved SMCGP program that accurately converges to e. Output from the evolved program is show in table 5. The program was first tested using an arbitrary precision maths utility to confirm the program is correct. At 2048 significant figures of precision, by iteration 100 the solution is correct to the first 121 digits, with an error of $4.386165e-122$.

At the top of the figure is seen a schematic of the evolved genotype. The node functions are written above the nodes. The initial genotype has 20 nodes. The next three graphs show the phenotypes at the first, second and third iterations. The ToDo list length was 2 so that only two SM functions are obeyed in each genotype/phenotype. The SM functions CHF and DU4 are used. The CHF function has the effect of changing the second COS function (at position 1) into DADD and the DU4 function results in nodes 1 to 3 being copied after node 3. Thus on each iteration the graph increases by three nodes: DADD, DADD and AVG respectively.

Consider the first iteration. The nodes from 4 to 22 form a block of nodes that merely shifts along at each iteration. We calculate what function these nodes calculate from an input x (supplied by the previous node). We have labeled the outputs of the nodes using x with the subscript being the same as node label. We obtain the following equations:

$x_4 = 2x$
$x_5 = 2x_4 = 4x$
$x_6 = x_5 = 4x$
$x_7 = x_6 x_6 = 16x^2$
$x_8 = x_5 + x_7 = 4x + 16x^2$
$x_9 = x_8 = 4x + 16x^2$
$x_{10} = \sqrt{x_8} = \sqrt{4x + 16x^2}$
$x_{11} = x_9 / x_7 = 1 + \frac{1}{4x}$
$x_{12} = {x_{11}}^{x_{10}} = (1 + \frac{1}{4x})^{\sqrt{4x+16x^2}}$
$x_{17} = x_{12} = x_{12}$
$x_{18} = x_{17} = x_{12}$
$x_{19} = x_{18} = (1 + \frac{1}{4x})^{\sqrt{4x+16x^2}}$

The value of x arises from the COS node at position 0 and the sequence of DADD, DADD, AVG nodes. The arguments of COS are zero so it outputs 1. The result of applying DADD, DADD, AVG is to produce the value 4. So at the first iteration $x = x_3 = 4$. Each iteration multiplies x by 4. Thus we can write $x = 4^{it}$ where it is the iteration. Clearly as the iterations increase x becomes very large. Defining $y = 4x = 4^{it+1}$ we can write the equation for the output x_{19} as

$$x_{19} = (1 + \frac{1}{y})^{y\sqrt{1+\frac{1}{y}}} \tag{2}$$

Eqn 2 tends to the form of a well-known Bernoulii formula (see above). Thus we have established mathematically that the evolved formula converges to e in the limit of large iterations.

7. CONCLUSIONS

SMCGP has been shown to successfully find good approximations to both π and e. The evolved approximation to e is a special case of previously known solution. We were unable to find a known solution similar to the found π approximation. We would speculate that this approach should be able to find novel approximations to both constants, however there is obvious difficulty in comparing a large number of generated solutions to a large body of known solutions.

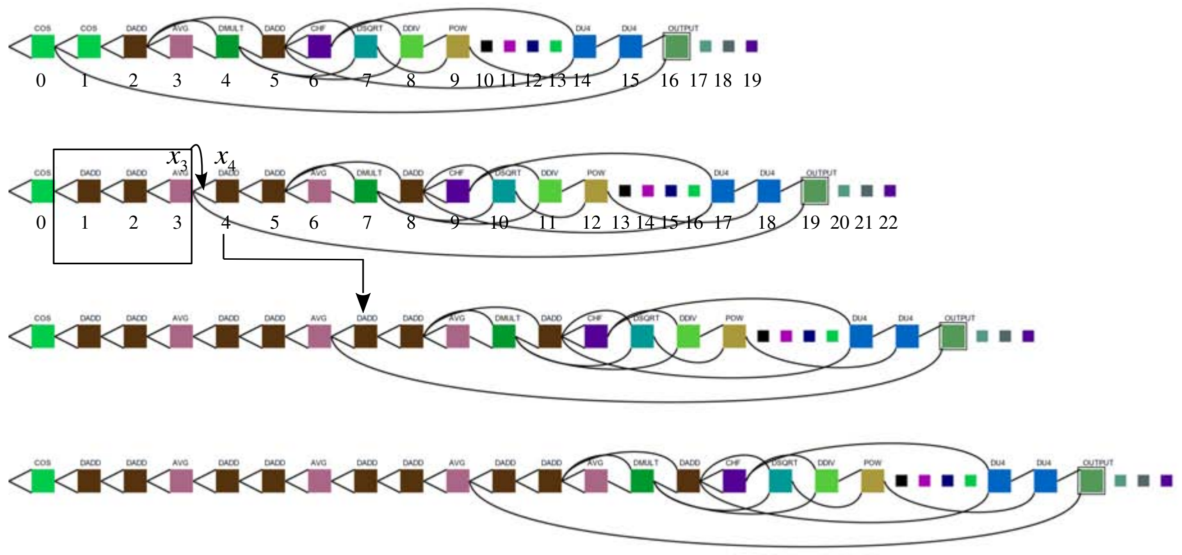

Figure 3: SMCGP genotype and three iterations of approximation to e

We also offer the problems of approximating to pi and e as benchmarks for developmental methods. Clearly they are difficult problems and also they are intrinsically interesting (the problem of finding e appears to be especially difficult).

Generating sequences of digits in a sequence such as π shows that SMCGP can generate programs that produce arbitrary sequences. We were mildly disappointed that we were unable to find a program that could generate the digits precisely, however the known solutions to this do not operate in base 10 [2]. Moving to different numeric representations may allow us to reproduce this type of result.

We also expect that SMCGP could be used to try to discover hitherto unknown but provable algorithms that sum mathematical series or converge to particular constants.

It was interesting to see that the self modifying aspects of SMCGP were used in both the solutions examined. In the approximation to e, the self modifying functions found a module that corresponded to a complicated power function. Conceivably such a function could have been evolved with a standard implementation of CGP, but we suspect that the self modification made it easier to discover iterative solutions. SMCGP can chose to ignore the SM functions, and revert in operation to CGP. If the SM was not useful then they should be ignored. We hope to investigate this in future work.

8. ACKNOWLEDGEMENTS

We would like to thank Jon Rowe for his valuable comments. WB acknowledges funding from Atlantic Canada's HPC network ACENET and by NSERC under the Discovery Grant Program RGPIN 283304-07.

[2]Such as the BBP Formula (described here: `http://mathworld.wolfram.com/BBPFormula.html`)that operates in base 16.

9. REFERENCES

[1] P. Bentley. Fractal proteins. *Genetic Programming and Evolvable Machines*, 5(1):71–101, Mar. 2004.

[2] H. J. Brothers and J. A. Knox. New closed-form approximations to the logarithmic constant e. *Mathematical Intelligencer*, 20(4):25–29, 1998.

[3] S. Harding, J. F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. In H. Lipson, editor, *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, pages 1021–1028. ACM, 2007.

[4] S. Harding, J. F. Miller, and W. Banzhaf. Evolution, development and learning with self modifying cartesian genetic programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 699–706, New York, NY, USA, 2009. ACM.

[5] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. In L. Vanneschi, S. Gustafson, et al., editors, *Genetic Programming, 12th European Conference, EuroGP 2009, Tübingen, Germany, April 15-17, 2009, Proceedings*, volume 5481 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2009.

[6] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Parity. In A. Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 285–292, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[7] J. Krohn, P. J. Bentley, and H. Shayani. The challenge of irrationality: fractal protein recipes for pi. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 715–722, New York, NY, USA, 2009. ACM.

[8] E. Maor. *e: The Story of a Number*. Princeton University Press, 1994.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, et al., editors, *Proc. of EuroGP 2000*, volume 1802 of *LNCS*, pages 121–132. Springer-Verlag, 2000.

[10] M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[11] M. D. Schmidt and H. Lipson. Solving iterated functions using genetic programming. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2149–2154. ACM, 2009.

[12] L. Spector, D. M. Clark, I. Lindsay, B. Barr, and J. Klein. Genetic programming for finite algebras. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298. ACM, 2008.

[13] M. Streeter and L. A. Becker. Automated discovery of numerical approximation formulae via genetic programming. *Genetic Programming and Evolvable Machines*, 4(3):255–286, 2003.