Symbolic Computation using Grammatical Evolution

Alan Christianson
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Rapid City, SD 57701
Alan.Christianson@mines.sdsmt.edu

Jeff McGough Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Rapid City, SD 57701
Jeff.McGough@sdsmt.edu

March 20, 2009

Abstract

Evolutionary Algorithms have demonstrated results in a vast array of optimization problems and are regularly employed in engineering design. However, many mathematical problems have shown traditional methods to be the more effective approach. In the case of symbolic computation, it may be difficult or not feasible to extend numerical approachs and thus leave the door open to other methods.

In this paper, we study the application of a grammar-based approach in Evolutionary Computing known as Grammatical Evolution (GE) to a selected problem in control theory. Grammatical evolution is a variation of genetic programming. GE does not operate directly on the expression but following a lead from nature indirectly through genome strings. These evolved strings are used to select production rules in a BNF grammar to generate algebraic expressions which are potential solutions to the problem at hand. Traditional approaches have been plagued by unrestrained expression growth, stagnation and lack of convergence. These are addressed by the more biologically realistic BNF representation and variations in the genetic operators.

1 Introduction

Control Theory is a well established field which concerns itself with the modeling and regulation of dynamical processes. The discipline brings robust mathematical tools to address many questions in process control. The mathematics are by no means a well defined list of equations to apply and for some problems, the tools are inadequate.

One of the basic questions in control theory is about the long term dynamics of a system under study. Many of the models of systems involve differential equations and some of these are nonlinear problems. The nonlinearity prevents us from obtaining an explicit analytic representation of the solution. Numerical methods are available and can provide solutions to a very high degree of accuracy. However, numerical approaches do not give a general qualitative description. So alternate methods are required to discuss what happens to the solution dynamics in a generic sense. One tool to address the problem is the Lyapunov function. It is a function which can shed some light on the system dynamics.

Although a Lyapunov function has a very precise definition, the definition is in no way constructive. The definition is more of a list of conditions which must be satisfied. To make matters worse, the list of conditions does not lead to a unique object. Several different functions may satisfy the definition of a Lyapunov function and all provide different information.

Our goal here is to illustrate the methods without getting mired down in excessive machinery required to treat general systems. So, we restrict the number of dependent variables to $N = 2$. The dynamical system we will focus on is

$$\frac{dx}{dt} = f(x, y), \quad \frac{dy}{dt} = g(x, y).$$

The functions f and g are assumed to be continuously differentiable in a open set $U \subset \mathbb{R}^2$. This assures us of local existence and uniquess of solutions; so we actually have solutions to discuss. The question then is given this system, can we predict the long term dynamics. In other words, what happens to $x(t)$ and $y(t)$ as $t \to \infty$. Assume for a moment that there was a point in U, (a, b), for which $f(a, b) = g(a, b) = 0$. This would be a point where all motion stopped. It is known as a rest point. We refine our question and ask if $x(t) \to a$ and $y(t) \to b$ as $t \to \infty$? This would tell us that the dynamics settles down to this rest point and thus tells us the behaviour of our system.

A candidate Lyapunov function is a function $L(x, y) : \mathbb{R}^2 \to \mathbb{R}$ which satisfies

- $L(a, b) = 0$,
- $L(x, y) > 0$ for $(x, y) \in U \setminus \{(a, b)\}$.

Simply stated this is a locally positive function away from the point (a, b). Lyapunov proved that if we also know that

- $\frac{\partial L}{\partial x} f(x, y) + \frac{\partial L}{\partial y} g(x, y) < 0$ for all $(x, y) \in U \setminus \{(a, b)\}$

then $x(t) \to a$ and $y(t) \to b$ as $t \to \infty$. This is known as stability. The point (a,b) is stable or it attracts the solution (sometimes called an attractor). The problem is simple. Find a function that satisfies the three conditions and we are done. We will have proved stability. Since Lyapunov gave conditions to satisfy, but not a constructive theorem, how can we find these functions?

It turns out to be very difficult in practice. Especially when in practice we are also interested in gaining the largest set U possible. This casts our problem as a search and optimization problem. For this we turn to evolutionary algorithms. They can act as both search and optimization approaches. There are different flavors of evolutionary algorithms and we choose a variation of one known as genetic programming.

Genetic Programming is a very successful and popular form of evolutionary computation. Koza and others (http://www.genetic-programming.org) have demonstrated the effectiveness of genetic programming in a diverse set of optimization problems such as circuit layout, antenna design, optical lens design, program generation and geometric optimization. Our interest is that genetic programming has been successfully applied in symbolic algebra problems. The Lyanpunov problem can be cast as an optimization problem involving symbolic algebra.

Genetic programs, like evolutionary algorithms in general, use the metaphor of biological evolution. First, it must represent candidate solutions in some encoded fashion. A collection of candidate solutions is normally generated; often done randomly. Continuing with the metaphor, a subpopulation is selected and two principle genetic operators are applied: that of mutation and recombination. This process of selection, mutation and recombination is repeated until a candidate solution satisfies the solution criterion. The specifics of selection, mutation and recombination, as well as additional operations on the population define the type of evolutionary algorithm. Pseudo-code for the approach is given in Figure 1. For more information on the breadth of the subject see [1, 3, 4].

A standard test problem in genetic programming is symbolic regression. GP is successful on the classical numerical regression problems even if it is not the fastest way to approach curve fitting. Unlike traditional regression where the function form is fixed and coefficients are desired, genetic programs may also be used to do symbolic regression which addresses the actual function form in addition to the coefficients. Since symbolic regression aims at producing a function which "fits" a data set, it opens up traditional areas of mathematics such as finding analytic solutions to differential equations [2, 5].

```
Initialize the population
Rank the population
Repeat until termination is met
   Select parents
   Recombination and mutation applied to produce new members
   Rank population
```

Figure 1: Evolutionary Algorithm

2 The Basic Genetic Program

A common approach in GP is to use an S-expression grammar for the storing of expressions. A valid S-expression is an identifier (a letter), a constant (floating point value), or an operation (a binary operation) along with its operands, which must be valid S-expressions themselves. To encode a quadratic:

$$ax^2 + bx + c \quad \rightarrow \quad (\,+\,c\,(\,*\,x\,(\,+\,b\,(\,*\,a\,x\,)))).$$

S-expressions are easily stored and manipulated as trees. Each leaf node corresponds to either an identifier or a constant, and the other nodes are binary operators. Evaluating an expression is simply evaluating the tree (depth-first algorithm to calculate the return value of the equation).

Given a specific quadratic, we can define a measure of how well this interpolates a data set by looking at the difference between the data points and the quadratic. For example, assume that we have the points (1,3), (4,7), (0,0), (-2, 5) and we have the polynomial $p(x) = x^2+1$. We compute

$$|p(1) - 3| + |p(4) - 7| + |p(0) - 0| + |p(-2) - 5| = |-1| + |10| + |1| + |0| = 12.$$

The smaller this error, the better the fit of the polynomial to the data set. In general this error may be written as

$$e = \sum_{i=0}^{N-1} |p(x_i) - y_i|$$

This sum is used to define the fitness function.

A population of possible functions may be formed. Using the error function, we can assign a fitness for each individual, with a smaller error corresponding to a higher fitness. This can be used to select subgroups of the overall population. Since the individuals are represented as trees, the genetic operators must act on the trees without destroying the tree. The mutation operator can be a simple random change of a node value or type. Recombination may act by exchanging randomly selected subtrees between two parent trees. By using the selection, mutation and recombination operations above we gain the basis for a genetic program.

Selection of parents may be done in a variety of ways. An effective approach is to randomly select parents from the population using a biased selection scheme where the bias is based on fitness. The process will normalize the fitness between 0 and 1. Next, generate a random number and randomly pick an individual. If the normalized fitness is above the random number, then keep the individual as a parent. After selection, the two genetic operators (recombination and mutation) are applied. For mutation, one may randomly select and change an element in the tree. For recombination, two parents are selected and in each a subtree is selected. These subtrees are exchanged. This process creates the new generation and we begin the cycle over.

This approach has been made popular by Koza, but sucessfully implemented by many authors. However, it is not without problems The first is that we don't seem to get convergence in the traditional sense due to stagnation. In this case, the population does not appear to be converging on a solution and seems to be stuck in the search. Another problem is that very large expressions are rapidly generated due to the nature of the recombination operator. These large expressions quickly come to dominate the population, resulting in extensive memory use and longer CPU times. Based on the reported problems found with classical GP, we decided to try a modification of the approach.

3 Grammatical Evolution

The central question remains: "can one evolve a Lyapunov Function"? Expression bloat was a serious problem and would cause the computation to stagnate due to excessive resource requirements; this problem in GP is noted in [7, 8] and it is suggested that the less destructive crossover operator in GE is responsible for the decrease in bloat when compared to similar implementations of GP. A different type of encoding scheme was explored to combat this problem. The new scheme was one designed to more closely model the biological systems on which evolutionary approaches are based. The physical expressions of traits, phenotypes, are not coded directly on the genome. The information on the genome, the genotype, is translated from DNA to RNA and then to proteins. These proteins will direct the construction of physical features. Very compact sequences can direct complicated structures due to the encoding. A similar approach is needed in genetic programming.

Grammatical evolution is an evolutionary algorithm which separates genotype and phenotype, allowing simple definition of the candidate solution and the flexibility to alter or replace the search algorithm. The genotype to phenotype translation, which is analogous to biological gene expression, is performed by interpreting a context free grammar stored in Backus Naur Form (BNF) and selecting production rules from that grammar based on the codons which make up the genotype. The use of a context free grammar to direct the creation of candidate solutions allows a greater level of control over their length and form than does traditional GP. The particular flavor of GE in use on this problem is driven by a steady state genetic algorithm using a variable length genome.

The steady state form of GA involves introducing only a single new individual in a given generation, unlike traditional generational models in which a significant portion of the population is replaced in a given generation. Steady state GAs have been found to outperform generational GAs in a number of applications, both with and without GE. This application also utilized a feature of GE known as wrapping, meaning that, upon reaching the end of the genotype, further production rules may be selected by starting over again at the first codon. This wrapping is implemented with a limit to avoid unbounded expansion of nonterminals in the grammar. This limits the amount of time spent in the gene expression as well as the maximum length of the resulting phenotype. The fitness determination is performed by direct evaluation of the expressions (as opposed to generating actual programs, compiling them, etc. as is often done with pure GP) using a free parser called fparser.

Although the decision to use GE was driven by the desire to control the candidate solutions in terms of both length and form, certain problems also require complicated crossover code to prevent the formation of nonsensical trees. The grammar-based construction of GE allows a simpler and finer control over candidate length and form. Domain specific knowledge can easily be included in the grammar to intelligently limit the solution search space.

A BNF grammar is one that is built from terminals, objects that are represented in the language such as constants, variables, and operators, and from non-terminals, objects that can be expanded into terminals or other non-terminals. It may be represented by the 4-tuple $\{N, T, P, S\}$, where for this application: $N = \{expr, op, preop, var\}$, is the set of non-terminals; $T = \{+, -, *, /, (,), X, 1.0, 2.0, \ldots, 9.0\}$ is the set of terminals; and $S \in N$, a start symbol; and P is the set of production rules, see Figure 2. Using the BNF grammar, an equation is formed from an integer string by applying the production rules which are selected based on the integer values.

```
(1) <expr> ::= <expr> <op> <expr>     (A)   (3) <pre-op>::= sqrt    (A)
             | ( <expr> <op> <expr> ) (B)
             | <pre-op> ( <expr> )    (C)   (4) <var>   ::= X       (A)
             | <var>                  (D)                 | 1.0     (B)
                                                          ....      ...
(2) <op>   ::= +                      (A)                 | 9.0     (J)
             | -                      (B)
             | *                      (C)
             | /                      (D)
```

Figure 2: Production rules

The rest of grammatical evolution is consistent with other forms of evolutionary algorithms. Algebraic expressions are represented via the string which the BNF grammar uses to create a individual as described above. A population is formed, often randomly to start, of n individuals. The normal process of selecting parents, crossover and mutation are applied to the population. Selection normally is some type of stochastic approach where the probability of selection is based on the fitness. This is used to produce a new generation. This process

is repeated until a candidate solution is found or the maximum generation count occurs.

Grammatical Evolution first distinguishes itself by the performance gains. GE on average uses significantly less memory than a comparable GP implementation and has a corresponding decrease in runtime compared to GP. These differences are due both to the compact binary representation of the genotype and the significant decrease in bloat of the resulting phenotype.

4 Algorithm details

The solution space is searched using a fixed size population approach. An initial population is generated by creating N vectors (individuals) of randomly-generated integers which serve as the codons of the individuals. They are replaced one at a time as individuals with higher fitness are created via crossover or other means, including the new initialization of individuals using the same technique as that used in the creation of the initial population.

As previously mentioned, evaluation of a candidate Lyapunov function $L(x, y)$ depends on the satisfaction of three conditions

- $L(a, b) = 0$,
- $L(x, y) > 0$ for $(x, y) \in U \setminus \{(a, b)\}$.
- $\dfrac{\partial L}{\partial x} f(x, y) + \dfrac{\partial L}{\partial y} g(x, y) < 0$ for all $(x, y) \in U \setminus \{(a, b)\}$

For a given system $F(x, y)$, $G(x, y)$ a region surrounding the rest point (a, b) is selected as the region of interest from which the points comprising the set U are selected. For this problem a user-defined grid is created, the points of which are used in the evaluation of all candidates in the population. For each point in U the three conditions are tested and a fixed penalty is applied for individuals which fail to satisfy any of them, rather than a variable penalty based on the degree to which the computed values vary from the desired values. The first two conditions are simple to evaluate; the third involves a finite difference approximation and thus requires additional work.

The partial derivatives of L with respect to x and y are approximated using finite differences. At each point (x, y), $L(x, y)$ is evaluated and the finite difference approximation is formed

$$\frac{\partial L}{\partial x} = \frac{L(x + \Delta x, y) - L(x, y)}{\Delta x},$$
$$\frac{\partial L}{\partial y} = \frac{L(x, y + \Delta y) - L(x, y)}{\Delta y}.$$

Normal usage is to take Δx to be the grid spacing, however, since we have the analytic expression, we can gain accuracy in taking Δx small, $\approx 10^{-6}$

For each condition, a fixed penalty of 1.0 is added for failure to satisfy the condition. For the third condition where the dot product is supposed to be non-positive, a smaller penalty of 1/3 is assessed for cases in which the dot product is approximately zero. This is because solutions were found where most of the evalated points were at this border condition instead of being strictly negative.

Although both generational and steady state models are available, the steady state loop is normally used. The goal is to keep it as close to the generational model as possible, in which G generations each result in the creation of N new individuals (minus some small retention from the previous generation...elitism etc).[1] For the steady state loop one individual in $N * G$ "generations" is created. Tournament selection is used to determine the parent population. N individuals are chosen randomly from the population and the best of these is selected as a parent. A random winner rate can be used to allow a randomly chosen individual to be used instead of the most fit individual from the tournament with the given probability. This is kept relatively low (.05 by default)

After selection, the genetic operators are applied. Early implementations mutated a single, randomly chosen codon in a specified individual. To increase population diversity, the currently used mutation operator iterates through each codon in an individual and replaces it with a randomly selected value with probability M. In other words, each codon has a 1 in 20 chance of being mutated if the crossover rate is .05, and it's possible but unlikely that every codon in a genotype could be mutated. This is relatively high incidence of mutation.

With mutation, recombination is also applied. Two parents are selected via the tournament selection, crossover is performed yielding two children. Using a variable length genome, a random cut point is selected such that it falls within the shorter of the two parents. The right halves of each parent are swapped, creating two new children. Both are evaluated and the more fit of the two is kept; The more fit of those is mutated and then introduced into the main population, replacing the least fit individual, unless it's a duplicate in terms of genotype. In the case of duplicates a brand new individual is created and introduced in it's place. This is done in an attempt to ward off stagnation and maintain diversity.

Once selection, mutation and recombination are completed, we have a new generation. This process repeats until the termination condition is met. This would be that a maximum number of steps has occured or that a solution with a particular fitness has been found.

[1] We use this term to connect to traditional genetic algorithms, but it can be misleading. For a steady state model the actual number population steps is $N * G$. It's a matter of semantics whether you consider each individual to be created in a new generation. Consider that one could create an individual, put it in the population, and then select that new individual as a parent for the next generation.

5 Results

A number of systems were used to test the ability of the software to find appropriate Lyapunov functions. For some systems and search regions there exist very simple Lyapunov functions of the form $x^2 + y^2$. Functions of this form are among the first guesses a human would make when trying to find a Lyapunov function by hand. In these cases, although the algorithm doesn't specifically look for the "obvious" candidates, it usually found these functions or variations on them in short order. Other systems presented more of a challenge, and it's in these cases that GE shined.

To simplify the problem and to direct the search in a reasonable direction a simple grammar to generate polynomials was used; see Figure 3. Typical parameter values are listed in Figure 4. Note on sampling radius and granularity: If the rest point is (0,0) and the radius is .3 with a granularity of .1, then the points sampled are (-.3, -.3), (-.3, -.2), ... (-.3, .3), (-.2, -.3)...(.3, .3).

```
<expr> ::= <expr><op><expr>
         | (<expr><op><expr>)
         | <var>^<const>
         | <const>*<expr>
<op>   ::= +
         | -
         | *

<var>  ::= x
         | y
<const> ::= 1
         | 2
         | 3
         | 4
```

Figure 3: Grammar for Lyapunov Functions

Population size (N): 200
Generation factor (G): 200
Crossover rate (C): .9
Mutation rate (M): .1
Generational model: steady state
Initial genotype length: 20 - 40 bytes
Tournament size (t): 5
Random tournament winner rate (w): .05
Sampling radius (r): .3
Sampling granularity (s): .1

Figure 4: Typical run parameters

The following provides some example systems and the highest fitness candidate Lyapunov functions. Plots are provided following the sample runs. The X in the plot is the rest point. The blue circles indicate grid points where the candidate Lyapunov function satisfies the conditions. The red squares are where the dot product is zero. These are often boundary cases.

5.1 Examples dynamical systems

1) For $f(x,y) = x(2-x-y), g(x,y) = y(3-x^2-y)$: Parameters: C = .9, M = .1, N = 200, G = 200. The restpoint is (2,0). The best two candidate Lyapunov functions are $L_1 = (1*(4*1*y^2+x^4+4*x^1)/x^3+y^4)$ and $L_2 = 2*y^2/x^3$, see Figure 5.

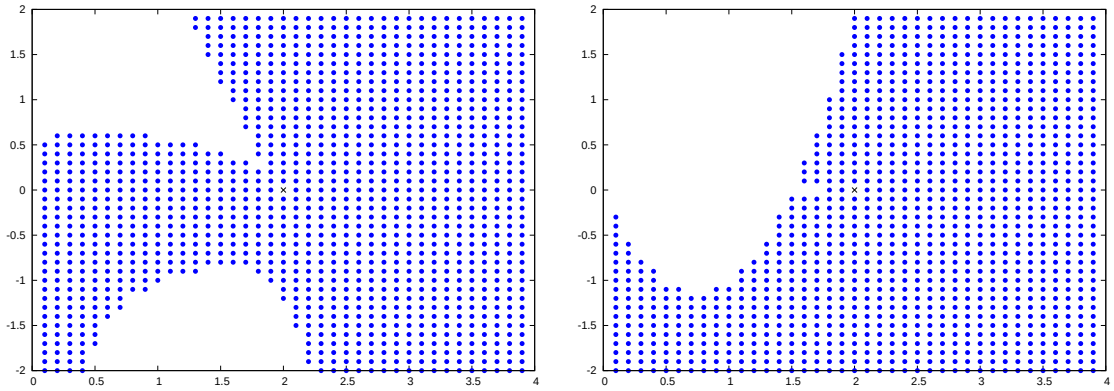

Figure 5: $f(x,y) = x(2-x-y), g(x,y) = y(3-x^2-y)$

2) For $f(x,y) = y, g(x,y) = -(x^2+y^2) - y/3$: Parameters: C = .9, M = .1, N = 200, G = 200. The restpoint is (0,0). The best candidate Lyapunov function is $L = (y^2 + (3*x^3 - 4*y^1)*x^3)$, see Figure 6.

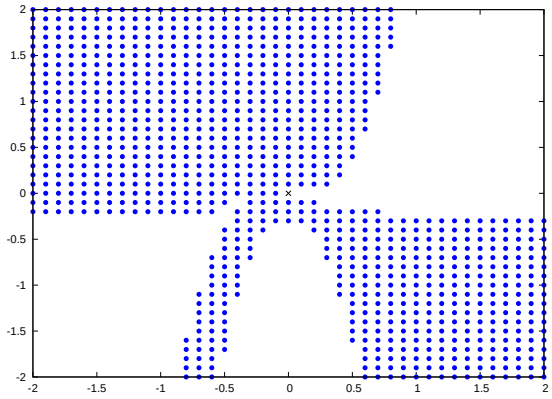

Figure 6: $f(x,y) = y, g(x,y) = -(x^2+y^2) - y/3$

3) For $f(x,y) = x-y-(x+y)^3, g(x,y) = -x-3*y-(x+y)^3$: Parameters: C = .9, M = .1, N = 200, G = 200. The restpoint is (0,0). The best candidate Lyapunov functions are $L_1 = y^4*3*y^2$, $L_2 = 4*2*(((4*4*2*x^4-x^4*4*2*y^4+y^4)-4*4*2*x^4-x^4)*4*2*y^4+y^4)$, $L_3 = ((4*x^3*2*x^1+3*y^2)*(4*y^1+4*x^3)*2*x^1+3*y^2)$. see Figure 7.

4) For $f(x,y) = (x+y)*(2-x)+(x-y)*(y-1), g(x,y) = (x+y)*(2-x)-(x-y)*(y-1)$: Parameters: C = .9, M = .1, N = 200, G = 200. The restpoint is (0,0). The best candidate

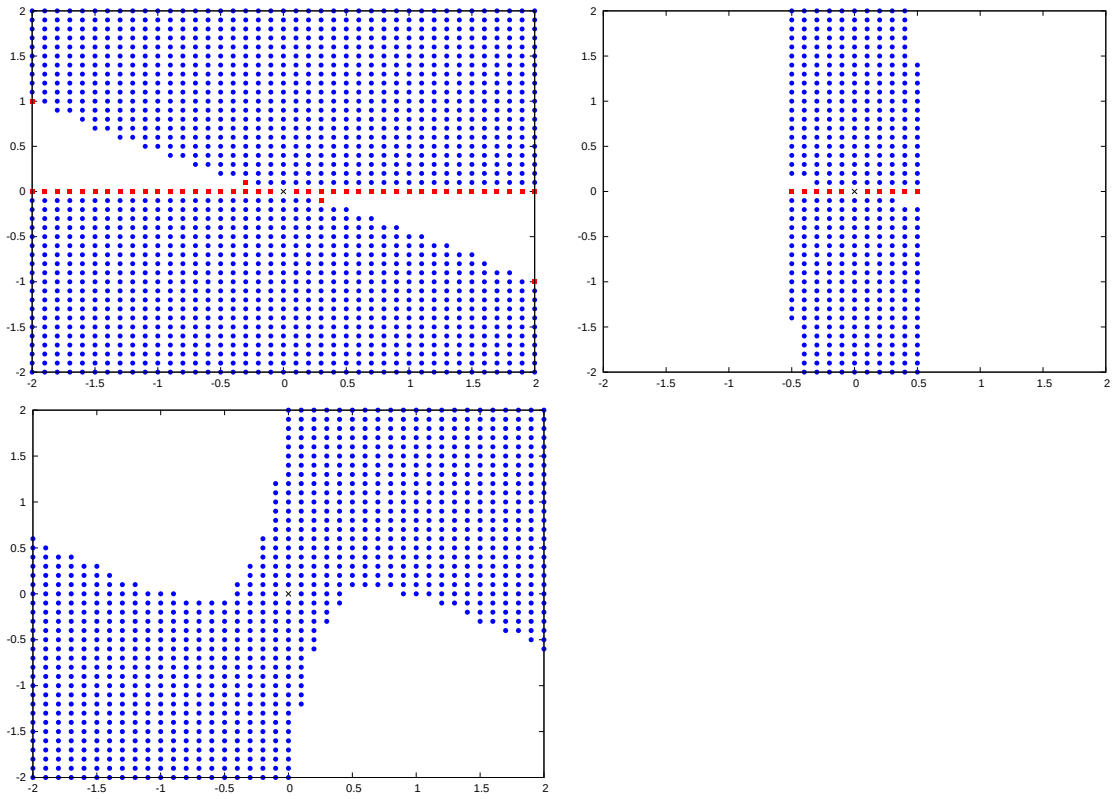

Figure 7: $f(x,y) = x - y - (x+y)^3, g(x,y) = -x - 3 * y - (x+y)^3$

Lyapunov function is

$$L = ((1 * (1 * (2 * (x^4 + (x^1 - (y^1 + y^2) + y^2)) * (x^4 + (x^1 - (y^1 + y^2) + y^2)))$$
$$*(x^4 + (x^1 - (y^1 + y^2) + y^2)) * (x^4 + (x^1 - (y^1 + y^2) + y^2)))$$
$$*(x^4 + (x^1 - (y^1 + y^2) + y^2))) * (x^4 + (x^1 - (y^1 + y^2) + y^2))),$$

see Figure 8. Note that this function may be written in a more compact form, but we choose to leave it in the same form as the algorithm produced it.

6 Conclusion and Future Directions

The injection of biological metaphors into the field of computer science has proved very valuable. A variation of genetic programming, Grammatical Evolution, is one such addition to the evolutionary computing toolkit. GE applied to the determination of Lyapunov functions has yielded Lyapunov functions which would be very difficult for an investigator to find. By simple modification of the fitness function, other applications in symbolic algebra can be treated.

The runtimes can be significant on these problems and use of a hybrid algorithm may be

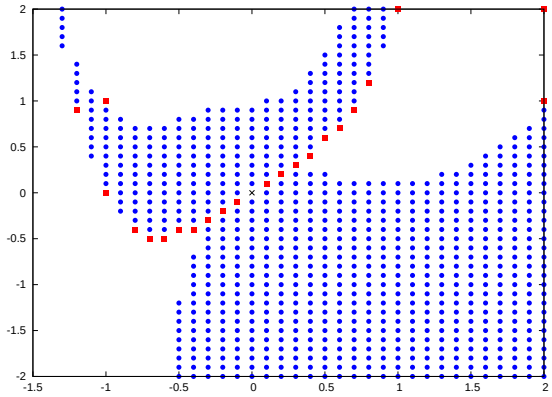

Figure 8: $f(x, y) = (x + y) * (2 - x) + (x - y) * (y - 1), g(x, y) = (x + y) * (2 - x) - (x - y) * (y - 1)$

warranted. The obvious step is to merge GE with differential evolution, known as grammatical differential evolution or GDE [6], which can tackle more difficult search spaces. Another approach is Local Search GE (LSGE) which adds a local search genetic operator such as hill-climbing (one approach is through the mutation operator).

References

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[2] Glenn Burgess. Finding approximate analytic solutions to differential equations using genetic programming. Technical Report DSTO-TR-0838, Surveillance Systems Division, Defence Science and Technology Organisation, Australia, Salisbury, SA, 5108, Austrlia, February 1999.

[3] David B. Fogel, editor. *Evolutionary Computation: the fossil record*. IEEE Press, Piscataway, NJ, 1998.

[4] David E. Goldberg and John H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2/3):95–100, 1988.

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

[6] M. O'Neill and A. Brabazon. Grammatical differential evolution. In *Proceedings of the ICAI 2006*, Las Vegas, Nevada, 2006. International Conference on Artificial Intelligence (ICAI'06), CSEA Press.

[7] Michael O'Neill and Conor Ryan. Under the hood of grammatical evolution. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99*, pages 1143–1148, San Francisco, CA, 1999. Morgan Kaufmann.

[8] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, 14-15 April 1998. Springer-Verlag.