SYMBOLIC COMPUTATION OF LYAPUNOV FUNCTIONS USING EVOLUTIONARY ALGORITHMS

Jeff McGough*, Alan Christianson, Randy Hoover
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Rapid City, SD USA

ABSTRACT

This paper is concerned with the question of stability in dynamical systems, specifically the issue of computing symbolic forms of Lyapunov functions for given dynamical systems. Due to the non-constructive form of the the Lyapunov constraints, we employ a type of evolutionary algorithm to construct candidate Lyapunov functions. Evolutionary Algorithms have demonstrated results in a vast array of optimization problems and are regularly employed in engineering design.

We study the application of a variant of Genetic Programming known as Grammatical Evolution (GE). GE distinguishes itself from more traditional forms of genetic programs in that it separates the internal representation of a potential solution from the actual target expression. Strings of integers are evolved, with the candidate expressions being generated by performing a mapping using a problem-specific grammar. Traditional approaches using Genetic Programming have been plagued by unrestrained expression growth, stagnation and lack of convergence. These are addressed by the more biologically realistic gene representation and variations in the genetic operators. Illustrative examples are presented to validate the proposed technique.

KEY WORDS
Lyapunov Functions, Grammatical Evolution, Evolutionary Computing, Symbolic Computation, Dynamical Systems.

1 Introduction

1.1 Overview

One of the fundamental questions that arises in nonlinear dynamical systems analysis is concerned with the stability properties of a rest point of the system. That is, given an n-dimensional dynamical system of the form

$$\dot{x} = f(x), \tag{1}$$

with $x(0) = x_0$, we have the following assumptions: 1) without loss of generality, we assume that $f(0) = 0$ is a rest point of the system, and 2) there exists a (sufficiently smooth) unique solution $x(t) \in \mathbb{R}^n$ in the open interval $U \subset \mathbb{R}^n$ that contains the rest point. The goal is then to develop a technique to understand the qualitative behavior of the rest point.

The typical technique used to achieve this goal is to use the theory of Lyapunov. If local behavior is desired Lyapunov's indirect method can be used to determine the behavior of the rest point by performing a linearization around the rest point and analyzing the stability of the linear system (techniques for doing this are well known). Unfortunately, this method has two fundamental drawbacks, 1) the results are only valid in a very local region of the rest point, i.e., U must be small and 2) if the linearized system has purely imaginary eigenvalues, the stability of the rest point is inconclusive. To overcome these drawbacks, we rely on the direct method of Lyapunov. The direct method of Lyapunov is formerly stated as follows [1]: Let $x = 0$ be a rest point of the system in (1), $U \subset \mathbb{R}^n$ be a domain containing $x = 0$, and $V(x) : U \to \mathbb{R}$ be a continuously differentiable function such that

$$\begin{array}{ll} (i) & V(0) = 0, \\ (ii) & V(x) > 0, \quad \forall x(t) \in U - \{0\}, \\ (iii) & \dot{V}(x) \leq 0, \quad \forall x(t) \in U - \{0\}, \end{array} \tag{2}$$

then the rest point $x = 0$ is stable. Furthermore, if $\dot{V}(x) < 0$ then the rest point $x = 0$ is asymptotically stable. The function $V(x)$ is commonly referred to as a Lyapunov candidate function.

Although the direct method of Lyapunov provides a technique to determine the qualitative behavior of a rest point of a dynamical system, the technique is not constructive, i.e., the theory gives no insight into how to actually construct $V(x)$. Furthermore, the conditions for stability are sufficient but by no means necessary. Therefore, if a given $V(x)$ only satisfies a subset of the properties outlined in (2), then the stability of the rest point is inconclusive and a new $V(x)$ need be constructed.

1.2 Related Work

The construction of Lyapunov candidate functions that will provide meaningful information about the rest point $x = 0$ leads to two questions, 1) can a simple procedure be developed to automate the construction of such functions and 2) can the Lyapunov functions constructed from this procedure provide information for a significant region of the domain $U \subset \mathbb{R}^n$. These two questions have led to a significant amount of research over the last several years. The naive approach to the construction of Lyapunov candidates

*Jeff.McGough@sdsmt.edu

is to attempt to evaluate the energy in the dynamical system [1, 2, 3]. If a good representation for energy can be obtained, this representation can sometimes be used as a Lyapunov candiate. Unfortunately, this technique rarely provides reliable results in practical applications. Another approach to constructing Lyapunov candidate functions is to use linear programming and a sum of squares decomposition [4, 5]. This approach relaxes the constraint that $V(x)$ satisfy the properties outlined in (2) and instead requires that both $V(x)$ and $\dot{V}(x)$ be sums of squares of polynomial functions. Some of the drawbacks associated with this technique however are that it requires that both $f(x)$ and $V(x)$ be polynomial functions. Again, in practice this may not be the case.

An alternate approach to constructing general Lyapunov candidate functions relies on evolutionary computing [6, 7, 8, 9]. The approach proceeds by utilizing techniques from genetic programming (GP). In particular, it has been shown that GP can be used to construct Lyapunov candidate functions by encoding a set of basis functions in a tree-like structure. An optimization is then performed over the domain U subject to the constraints outlined in (2). The fundamental drawbacks associated with using GP for the construction of Lyapunov candidates are that the resulting expression for $V(x)$ produced by the GP may be extremely large (this is referred to as expression bloating), as a result, convergence to a solution may be slow and/or nonexistent (refer to Section 2).

The current work is also motivated by evolutionary computing, however, to improve on the performance and usability shown by previous GP-based approaches, we propose using grammatical evolution (GE). While both GP and GE are forms of evolutionary algorithms, the manner in which they represent or encode the solution is actually quite different (refer to Section 3). It has been shown that for some applications, GE is capable of outperforming GP in terms of convergence speed, storage requirements and solution structure [10, 11, 12]. Specifically, GE typically does not suffer from expression bloating like GP does, thereby significantly improving the convergence rate and reducing storage costs. Furthermore, the resulting analytic solutions are often smaller and significantly less complicated.

The remainder of this paper is organized as follows: In Section 2 we outline the basics of a genetic program and discuss some of the advantages and drawbacks associated with using GP for symbolic regression. In Section 3 an introduction to grammatical evolution is presented along with a typical example set of production rules. The algorithmic details used in the current work are then presented in Section 4. Several illustrative examples are then presented in Section 5 with conclusions and future directions outlined in Section 6.

2 The Basic Genetic Program

2.1 Evolutionary Computing

Genetic Programming is a very successful and popular form of evolutionary computation. Koza *et al.* [13] have demonstrated the effectiveness of genetic programming in a diverse set of optimization problems. Specific examples include circuit layout, antenna design, optical lens design, program generation, geometric optimization, and symbolic regression. To illustrate the capabilities of GP, several authors have studied the idea of using genetic programming to find Lyapunov functions [7, 8, 9, 14].

Genetic programs, like evolutionary algorithms in general, use the metaphor of biological evolution. First, as with all optimization algorithms, candidate or approximate solutions are encoded in some fashion, using an array or a string to represent the candidate. In evolutionary computing, the encoded solution is called the genome and is a digital counterpart to its biological analog, DNA. The candidate solution is also referred to as an individual. A collection of individuals or candidate solutions is called the population. To start we normally create an individual by filling out the genome with random values. The initial population is formed by taking n of the "random" individuals.

As a simple example, take the design of a circuit which uses two resistors, two capacitors and one inductor. If the structure of the circuit is fixed, we may encode the circuit using an array $v = \{r_1, r_2, c_1, c_2, l_1\}$. The array represents the genome and a specific array $\{1500, 2200, 10, 100, 1\}$ is thought of as an individual. A starting population will have n of these arrays, all filled with randomly generated values.

Continuing with the metaphor, parents are chosen from the population and an offspring is produced from the application of two principle genetic operators: mutation and recombination (or crossover). Next, selection is applied to see which members survive or are removed. Selection of parents and selection of the surviving members is done using a fitness or cost function. In the case of a maximization problem, the fitness is directly related to the cost and more fit individuals are selected for reproduction and survival.

This process of parent selection, reproduction and survival is repeated until a candidate solution satisfies the solution criterion. One expects that each new generation will have more highly fit individuals and thus the average individual (or population average) is improving. The specifics of this process, such as how the solutions are encoded, how the selections take place, and how reproduction is performed define the flavor of evolutionary algorithm one is using. Many of the approaches in the literature follow a biological model which allows intuition to aid in algorithm development. Pseudo-code for the approach is given in Figure 1. For more information on the breadth of the subject see [15, 16, 17].

The best known of the evolutionary algorithms is known as Genetic Algorithms. The genomes are bit strings which are then evolved using the two genetic operators mentioned above. A simple extension is to use an array of floats instead of bits. Using the array of floats, we may store the coefficients of a polynomial:

$$p(x) = a_5x^5 + a_4x^4 + a_3x^3 + x_2x^2 + x_1x + a_0$$

$$\rightarrow [a_5, a_4, a_3, a_2, a_1, a_0].$$

The genetic operators are straighforward. Randomly select an index, $rand()\%5 = i$, and replace $a_i \rightarrow (1 + p) * a_i$

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

Figure 1. Evolutionary Algorithm

(where p is drawn from some distribution). For example, if $i = 4$, $p = 1/2$ and the genome is $[1, -2, 4, 0, -2, 3]$, the string changes to $[1, -2, 4, 0, -3, 3]$. Recombination is done by selecting two parents from the parent population and randomly selecting a cut point in the array. Then we cut and swap. This is best understood with an illustration. If dad is $[a_5, a_4, a_3, a_2, a_1, a_0]$, mom is $[b_5, b_4, b_3, b_2, b_1, b_0]$ and the cut point is $i = 2$, then the child is $[a_5, a_4, b_3, b_2, b_1, b_0]$.

A two variable version of this approach succeeds at finding candidate Lyapunov functions [18]. One may select any number of functions to work with, so the basis is quite flexible here. In the higher dimensional cases the genome is an array, P with $x \in \mathbb{R}^n$ and the candidate is then $V = x^t P x$. However, the drawback is that the function form must be given or known apriori. Really novel solutions do not arise in this approach. Novel solutions are a strength of evolutionary computation in general and so more robust approaches are warranted.

2.2 Genetic Programming

A standard test problem in genetic programming is symbolic regression and is closely related to the process of determining Lyapunov functions. Koza [19] has demonstrated that GP is successful on classical numerical regression problems, however, as you might expect, it is not a particularly fast (or efficient) approach to curve fitting. Unlike traditional regression where the function form is fixed and coefficients are desired, genetic programs may also be used to determine function form along with the coefficients.

A common approach in GP is to use an S-expression grammar for the storing of expressions. To encode a quadratic we rewrite it as

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

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.

A population of possible functions may be formed, and an error function can be used to assign a fitness to each individual, where small error corresponds to a higher fitness. This fitness level can then be used to select subgroups of the overall population to generate offspring. Because the individuals are represented as trees, the genetic operators must act on the trees without destroying them. The mutation operator can be a simple random change of a node value or type, and 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.

One of the interesting aspects of evolutionary algorithms is in tuning them for a particular problem and one of the first decisions to be made is in the selection of the parent population. Selection of parents may be done in a variety of ways. A simple choice is to sort the population according to fitness and take the top k%. Normally the population size is kept constant so the number of offspring will balance the number eliminated. It is up to the algorithm author to decide how the sample population is controlled.

Using a strict (or deterministic) ranking process tends to produce results like gradient methods and local extremals can trap progress. To avoid the local extremal traps, stochastic or non-deterministic approachs are used. After selection, the two genetic operators (recombination and mutation) are applied. The mutation operator is normally applied to the offspring, but can be applied to random members of the population. The probability of a mutation event is a selectable parameter, as are the number of mutation events per genome. The location of the mutation on the genome is normally randomly selected as well. For GP specifically, mutation can be applied to any node in the tree and can be applied multiple times at multiple locations. A similar decision process must be made for recombination. For recombination in GP, two parents are selected and a subtree from each parent is exchanged. This process generates a new generation and the cycle is then repeated.

The idea behind GP was made popular by Koza, but has been successfully implemented by many authors. Applying GP to the construction of Lyapunov candidates provides an attractive solution, however, a few problems arise. One problem is known as expression bloat. Recombination with tree data structures results in very large expressions. The problem arises due to non-uniform cutting and pasting so that one offspring gains a much deeper tree. Unrestrained tree growth is an intrinsic aspect to GP. Large trees take up significant memory and CPU resources which results in performance degradation, long run times, and few useful resulting functions. Our attempts to resolve this by placing penalties for expression size have not proved successful. A second problem we found was stagnation. Like the numerical counterparts, the GP could keep running and never converge (in the sense here of having a member of the population which satisfied a given constraint set).

Based on these problems, we are interested in a type of evolutionary algorithm that had finite genome size and fast convergence. The answer came from going back to the biology and adding a layer in the process.

3 Grammatical Evolution

The physical expressions of traits or biological forms, known as phenotypes, are not coded directly on the genome

(the DNA). The information on the genome, known as the genotype, is translated from DNA to RNA and then to proteins. These proteins will direct the construction of physical features. Thus we witness in the cell very compact sequences that can direct complicated structures due to the encoding. A similar approach is desired in evolutionary algorithms; one designed to more closely model the biological systems on which evolutionary approaches are based.

Grammatical evolution is an evolutionary algorithm which separates the encoding from the final form, in other words separates the genotype and phenotype. This allows a simple definition of the candidate solution and the flexibility to alter or replace the search algorithm - a type of modularity. This is the most significant difference between GE and GP because GP does not directly encode structure (phenotype).

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. 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 Table 1. 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.

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 grammar provides a middle layer in the chain from genome string to symbolic function, e.g.,

$$\boxed{\text{Genome}} \rightarrow \boxed{\text{BNF Grammar}} \rightarrow \boxed{\text{Lyapunov Function}}.$$

The particular flavor of GE in use on the Lyapunov problem is driven by a steady state evolutionary algorithm using a variable length genome. The steady state form 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 genetic algorithms (GAs) have been found to outperform generational GAs in a number of applications, both with and without GE [20]. Our approach 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.

Expression bloat associated with GP is a serious problem in that it has a tendency to cause the search for an optimal solution to stagnate due to excessive resource requirements. It has been shown however that for similar implementations, GE does not typically suffer from expression bloat as compared to GP due to the less destructive recombination operation [10, 21]. Grammatical Evolution at 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.

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.

Table 1 provides an example of genotype to phenotype mapping. Table 1(a) shows the grammar, (b) shows the genome and (b) shows each non-terminal being translated using the grammar. The process begins with the start symbol being mapped to <expr>. Beginning at the first codon in the genotype, a rule is selected to replace the first non-terminal in the expression with something else. The codon's value (4) maps to zero when taken modulo four, which is the number of rules for the non-terminal. The expression is expanded to <expr><op><expr>. The second codon is used to evaluate the first non-terminal in the next expression. This process repeats as shown, until the final expression $2 + (X * 4)$ is found. This represents the phenotype of the individual and is used in the fitness function to evaluate and rank the individual.

Other than the BNF encoding scheme, much of GE is consistent with other forms of evolutionary algorithms. An initial population of individuals is created by generating vectors of random integers to represent each individual's genotype. The normal process takes place: selecting parents, performing crossover and mutation to produce offspring, and replacing individuals in the current population with the newly created individuals. A tournament selection process is used, in which T individuals are randomly selected from the population. From each set of T individuals, one is chosen to be a parent. The choice is a biased random selection, with more fit individuals having a greater probability of being chosen. An alternate approach is used for the latter selection, with a randomly chosen individual being selected with probability W (typically a small value such as .05) and the most fit individual from the subset being selected with probability $1 - W$. The process continues until either the maximum number of generations are formed or an acceptably-fit individual is found.

4 Lyapunov Functions Using Grammatical Evolution

4.1 Problem Statement

Our objective is to construct a function $V(x)$ such that the conditions outlined in (2) are satisfied. To this end, we

(a)

```
A) <expr> ::= <expr><op><expr>  (0)
            | (<expr><op><expr>) (1)
            | <const>           (2)
            | <var>             (3)

B) <op> ::= +  (0)
          | -  (1)
          | *  (2)
          | /  (3)

C) <var> ::= X (0)

D) <const> ::= 1 (0)
             | 2  (1)
             | 3  (2)
             | 4  (3)
```

(b)

4	10	1	8	5	3	17	2	6	3

(c)

Codon	Rule	Replacement	Expression
4	4%4 = 0	<expr>⇒ <expr><op><expr>	<expr><op><expr>
10	10%4 = 2	<expr>⇒ <const>	<const><op><expr>
1	1%4 = 1	<const>⇒ 2	2 <op><expr>
8	8%4 = 0	<op>⇒ +	2 + <expr>
5	5%4 = 1	<expr>⇒ (<expr><op><expr>)	2 + (<expr><op><expr>)
3	3%4 = 3	<expr>⇒ <var>	2 + (<var><op><expr>)
17	17%1 = 0	<var>⇒ X	2 + (X <op><expr>)
2	2%4 = 2	<op>⇒ *	2 + (X * <expr>)
6	6%4 = 2	<expr>⇒ <const>	2 + (X * <const>)
3	3%4 = 3	<const>⇒ 4	2 + (X * 4)

Table 1. An annotated grammar showing production rules with their associated indices. The non-terminals are labeled A-D, and each production rule is shown with its numeric offset, from 0 to N-1, where N is the number of rules for a given non-terminal.

cast this as a constrained optimization problem as follows: Given a set $D \subset \mathbb{R}^n$, find some $V(x)$ such that we

Maximize: $U \subset D$;

Subject to:

$$\begin{array}{lll} (i) & V(0) = 0, & \\ (ii) & V(x) > 0, & \forall x(t) \in U - \{0\}, \\ (iii) & \dot{V}(x) \leq 0, & \forall x(t) \in U - \{0\}. \end{array}$$

This recasting transforms the problem of constructing the Lyapunov candidate function into a symbolic regression problem. As such, as discussed in Section 3, grammatical evolution is an ideal candidate for determining a solution to such problems. The difficulty lies in the evaluation of the fitness function. We begin the process in the same manner as any evolutionary algorithm.

4.2 Algorithm Construction

The candidate function space is searched using a fixed size population approach. An initial population is generated by creating n vectors of randomly-generated integers which serve as the genotypes of the individuals. The individuals are replaced one at a time as individuals with higher fitness are created via crossover or mutation. The algorithm seeks to discover the most fit individual, which in this case means minimizing the fitness value.

The challenge is to create a meaningful fitness function. Although we are evolving symbolic formulas, we do not perform symbolic algebra and calculus. Thus checking conditions (*ii*) and (*iii*) are done numerically. For a given system, $\dot{x} = f(x)$, a region surrounding the rest point $x = 0$ is selected as the sample region (from which the points comprising the set D are selected). A user-defined grid is created to evaluate conditions (*i*) - (*iii*). In the current work

$$D = \{x_i : \|x_i\|_\infty < r\}.$$

Over the sample set D, we can test condition (*ii*) at each sample point[1]. Condition (*iii*), the computation of $\dot{V}(x)$, requires further elaboration. First, we note that

$$\dot{V}(x) = \nabla V(x) \cdot f(x) = L_f V(x),$$

where $L_f V(x)$ is the Lie derivative of $V(x)$. The partial derivatives are estimated using a three point central finite difference approximation. This allows $L_f V(x)$ to be evaluated at each sample point in D.

The fitness is then evaluated by verifying that condition (*i*) holds and that conditions (*ii*) - (*iii*) are satisfied for all samples $x_i \in D - \{0\}$. These criteria are tested and penalties are assessed, lowering the fitness when a criterion is not met. If $V(0) \neq 0$ then a penalty of 100 is assessed, $V(x) \leq 0$ is assessed a penality of 1.0, $\dot{V}(x) > 0$ assessed 1.0 and $\dot{V}(x) = 0$ assessed 0.33.

The design decision to use fixed values is based on the goal of identifying the largest set $U \subset D$ in which conditions (*i*) - (*iii*) hold. Though an interesting alternative, a variable, scaled penalty has not been investigated. Undefined operations such as division by zero cause a large penalty (300.0) to be assessed. In some tests a protected division operator (common in genetic programming) was used to prevent exceptions in the evaluation of expressions. This operator returns 1 when a division by zero occurs but it was rarely needed.

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).[2] For the steady state loop, one individual is created in each of $N * G$ "generations". Tournament selection is used to determine the parent population. N individuals are chosen randomly from

[1]The first condition can be ensured by adjusting a candidate $V(x)$ in the case that $V(x = 0) = Z$. We instead use $V(x) - Z$ as our candidate to ensure condition (*i*) is satisfied.

[2]We use this term to connect to traditional genetic algorithms, but it can be misleading. For a steady state model the actual number of 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.

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 which serves to balance other parameters which are more elitist and tend towards stagnation and settling in local minima.

With mutation, recombination is also applied. Two parents are chosen via tournament selection and 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 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 its 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 occurred or that a solution with a particular fitness has been found.

5 Illustrative Examples

5.1 Overview

A number of systems were used to test the ability of the proposed approach to construct appropriate Lyapunov functions. Although the above development is capable of operating in $\mathbb{R}^n$, for easy visualization of the phase-space, we restrict our attention in this section to $\mathbb{R}^2$. Therefore, we consider the system $\dot{x}_i = f_i(x_i)$ where $i = 1, 2$. Furthermore, without loss of generality, we assume that $x = 0$ is a rest point of the dynamical system.

For most dynamical systems, the initial Lyapunov candidate "guess" typically results in the Lyapunov candidate function being some quadratic form, i.e., $V(x) = x^T P x$ for some $P > 0$. We show that for dynamical systems in which this "guess" is valid, the algorithm usually finds these "obvious" candidates in short order even though it doesn't have any specific knowledge about these forms *a priori*. For other systems however, constructing a suitable Lyapunov candidate function proves to be significantly more challenging. It is these particular systems in which the application of GE shines.

Different grammars were used, depending on the system being investigated. Grammar tuning is an important part of any application of GE, as the grammar defines the search space and allows the user to direct the search using domain-specific knowledge. Small changes in grammar can have a dramatic effect on the success of a search. To illustrate this behavior, a number of runs were completed using the grammar shown in Figure 2, then repeated with `<var>^<const>` being replaced by `<var>^<expr>`. Though a small change in terms of the grammar definition, expressions generated using the original grammar are restricted to exponents of 1,2,3, or 4. The modified grammar allows all manner of expressions to serve as the exponent for a variable, resulting in many expressions which are unsuitable for the problem domain thereby decreasing the success rate and mean fitness.

```
<expr> ::= <expr><op><expr>
         | (<expr><op><expr>)
         | <var>^<const>
         | <const>*<expr>
<op>   ::= +
         | -
         | *
<var>  ::= x
         | y
<const> ::= 1
         | 2
         | 3
         | 4
```

Figure 2. Grammar 1: A basic polynomial grammar

Population size (N)	200
Generation factor (G)	50
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)	1.0
Sampling granularity (s)	.1
Protected divide	Disabled
Penalty for $\dot{V}(x) = 0$	0.33

Table 2. Typical run parameters

The following subsection provides some example dynamical systems along with their resulting highest fit Lyapunov candidate function. For each example, plots of the phase-space are provided depicting the the sample grid points $x_i \in D$ (denoted by blue circles) where the current Lyapunov candidate function is evaluated. In each system, the grid size r is set to $1, 2$ depending on the system of under consideration. The red circles in each figure denote regions in $U \subset D$ are where the Lie derivative $L_f V(x) \equiv 0$. The standard run parameters for each example system are outlined in Table 2. For completeness, the candidate functions are reported in the same form as the run generated and no algebraic simplification has been done (we leave $\sqrt{x^2}$ instead of rewriting as $|x|$).

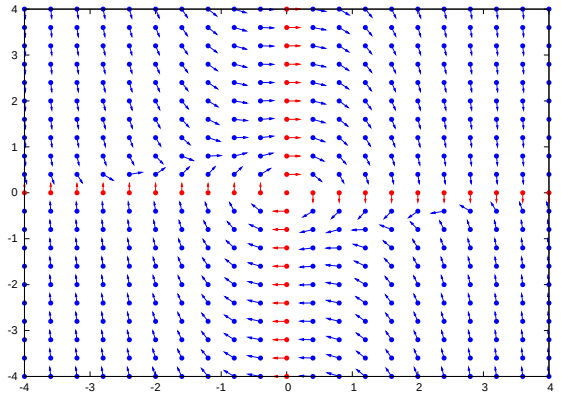

Figure 3. Results for $f_1 = x_2, f_2 = -x_1(1 + x_1x_2)$

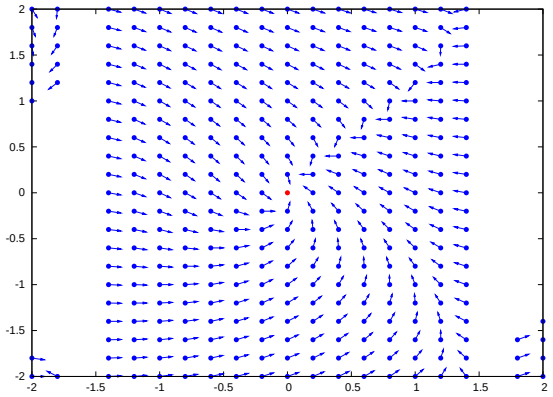

Figure 4. Results for $f_1 = -\tan(x_1) + x_2^2, f_2 = -x_2 + x_1$

5.2 Examples of dynamical systems

5.2.1 Example 1:

In the first example, we consider the system

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_1(1 + x_1x_2), \end{aligned}$$

the results of which are depicted in Fig. 3. This is a typical example in which the evolved function is the standard guess $V(x) = x_1^2 + x_2^2$, which satisfies the conditions of stability ($L_fV(x) = 0$) for some $U \subset D$. In particular, for this example,

$$L_fV(x) = -2x_1^2x_2^2.$$

The evolved candidate function is our standard guess and a weak Lyapunov function. Stability will follow from an application of LaSalle's theorem [1]. The population size was 100 with 50 generations.

5.2.2 Example 2:

For the second example, we look at the equations

$$\begin{aligned} \dot{x}_1 &= -\tan(x_1) + x_2^2, \\ \dot{x}_2 &= -x_2 + x_1, \end{aligned}$$

the results of which are depicted in Fig. 4. This example shows the valid domain for the candidate function $V = x_1^2 + (x_2^2\sqrt{x_2^2})$. This evolved function provides an example of evolutionary novelty by creating an absolute value function $\sqrt{x_2^2}$ used in the candidate. The population size was 100, the number of generations was 100 and a lower mutation rate of 0.05.

5.2.3 Example 3:

For the last example, we look at the equation of the damped pendulum

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\sin(x_1) - x_2, \end{aligned}$$

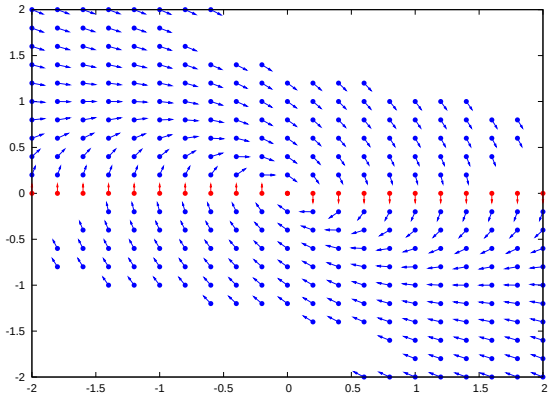

Figure 5. Results for $f_1 = x_2, f_2 = -\sin(x_1) - x_2$

the results of which are depicted in Fig. 5. This system has a stable (spiral) rest point at the origin. One of the evolved candidate Lyapunov functions for this example is $V(x) = x_2^2 + 4\sin(x_2^2) + 4x_1^2$. The standard Lyapunov candidate "guess" for this system is computed by looking at both the kinetic and potential energy of the system, resulting in $V(x) = \frac{x_2^2}{2} + (1 - \cos(x_1))$.

We see the stability region extending farther out than $E = x_1^2 + x_2^2$ would provide since adjacent rest points will prevent E from being a global Lyapunov function. Note that the function V is actually a weak Lyapunov function. However in this case, the flow is tangent to the manifold for which $\dot{V} = 0$ and a stability argument follows. This example made one of the more interesting plots.

6 Conclusion and Future Directions

In this paper we introduced a biologically motivated search process known as evolutionary computing. We were able to show that a variation of genetic programming, Grammatical Evolution, can find candidate Lyapunov functions. GE applied to the determination of Lyapunov functions has yielded both the obvious "energy" functions and some functions that were completely novel. We have found that approach gives true Lyapunov functions and functions useful in the application of Lasalle's invariance. Although

each candidate function is useful on its own, it also is useful in combination with other possible known Lyapunov functions; extending the reach of traditional functions like the energy functions.

Future work will include using this technique for invariant regions. We feel that this tool will aid in addressing the problem of finding maximal invariant sets. We anticipate looking at systems with limit cycles and more complicated phase portraits. It will also be useful to extend to higher dimensions to see how the GE approach scales. One last possibility is to add grammatical differential evolution to improve the search [22].

References

[1] Hassan K. Khalil, *Nonlinear Systems*, Prentice Hall, New Jersy, 2002.

[2] Horacio J. Marquez, *Nonlinear Control Systems: Analysis and Design*, Wiley, New Jersy, 2003.

[3] Shankar Sastry, *Nonlinear Systems: Analysis, Stability, and Control*, Springer, New York, 1999.

[4] A. Papachristodoulou and S. Prajna, "On the Construction of Lyapunov Functions using the Sum of Squares Decomposition", in *IEEE Conf. on Dec. and Cont. (CDC)*, 2002.

[5] S. Prajna, A. Papachristodoulou, and F. Wu, "Nonlinear control synthesis by sum of squares optimization: a lyapunov-based approach", in *Asian Conf. Cont. (ASCC)*, 2004.

[6] B. Grosman and D. R. Lewin, "Automatic Generation of Lyapunov Functions Using Genetic programming", in *Int. Fed. Automat. Cont. (IFAC)*, 2004.

[7] B. Grosman and D. R. Lewin, "Lyapunov-based stability analysis automated by genetic programming", in *IEEE International Symposium on Computer-Aided Control Systems Design, 2006*, Munich, Germany, 4-6 Oct. 2006, pp. 766–771, IEEE.

[8] Benyamin Grosman and Daniel R. Lewin, "Lyapunov-based stability analysis automated by genetic programming", *Automatica*, vol. 45, no. 1, pp. 252–256, 2009.

[9] Kuan Luen Ng and Rolf Johansson, "Evolving programs and solutions using genetic programming with application to learning and adaptive control", *Journal of Intelligent and Robotic Systems*, vol. 35, no. 3, pp. 289–307, Nov. 2002.

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

[11] Conor Ryan and Michael O'Neill, "Grammatical evolution: A steady state approach", in *Late Breaking Papers at the Genetic Programming 1998 Conference*, John R. Koza, Ed., University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998, Stanford University Bookstore.

[12] Michael O'Neill, Conor Ryan, Maarten Keijzer, and Mike Cattolico, "Crossover in grammatical evolution: The search continues", in *Genetic Programming: 4th European conference*, Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, Eds., Berlin, 2001, pp. 337–347, Springer.

[13] John R. Koza, "www.genetic-programming.org", 2010.

[14] T. Tsuzuki, K. Kuwada, and Y. Yamashita, "Searching for control lyapunov-morse functions using genetic programming for global asymptotic stabilization of nonlinear systems", in *45th IEEE Conference on Decision and Control*, San Diego, USA, 13-15 Dec. 2006, pp. 5114–5119, IEEE.

[15] 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, Jan. 1998.

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

[17] David E. Goldberg and John H. Holland, "Genetic algorithms and machine learning", *Machine Learning*, vol. 3, no. 2/3, pp. 95–100, 1988.

[18] Carin Smith, "Evolving Lyapunov Functions", Tech. Rep., South Dakota School of Mines and Technology, Rapid City, SD, USA, May 2006.

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

[20] Alex Rogers and Adam Prügel-Bennett, "Modelling the dynamics of a steady-state genetic algorithm", in *Foundations of Genetic Algorithms (FOGA-5). Preliminary Version of the Proceedings*, pp. 161–171. Leiden, 1998.

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

[22] 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.