Searching for Lyapunov Functions using Genetic Programming

Carl Banks
Virginia Polytechnic Institute and State University, Blacksburg, VA 24060

There is currently no generally-applicable way to find Lyapunov functions for stable nonlinear systems in a reasonable amount of time. However, genetic programming is exciting, new possibility. Genetic programming is a variation of genetic algorithms where the objective space is a space of heirarchial tree structures. The tree structures, among other things, can represnt represent almost any mathematical expression. I have implemented a genetic programming algorithm, in Mathematica, which searches for a Lyapunov function of a given system, where the tree structures represent potential Lyapunov functions. The implementation evaluates the "Lyapunovness" of the functions by testing the Lyapunov conditions ($V(x) > 0$ and $\dot{V}(x) \leq 0$) at many random points. The implementation was successful somewhat in finding Lypunov functions for simple, two-dimensional systems.

Introduction

Lyapunov functions are of fundamental importance in the study of nonlinear systems. Not only are Lyapunov function useful in proving stability of nonlinear systems, they are an important part of certain nonlinear control design methods, including backstepping and sliding mode.

Unfortunately, there are no general methods for finding a Lyapunov function for a given system. In this report, I apply genetic programming, a heuristic optimization technique that is a variation of genetic algorithms where the objective vectors can be complex mathematical expressions, to the general Lyapunov-finding problem.

Genetic programming is a relatively new field, first introduced by Koza in 1992[1]. Ref. 1 remains the most comprehensive introduction and reference for genetic programming. As yet, there is very little literature on the application of genetic programming to nonlinear dynamics problems. Of the few papers that examine this, most focused on system identification[2] or controller design by searching for a control law[3].

This report first reviews genetic algorithms and genetic programming, before examining the Lyapunov search algorithm.

Genetic Algorithms

A genetic algorithm is a heuristic optimization method based on the idea of breeding*. A dog breeder will choose dogs with desirable characteristics (largeness, smallness, spottedness, quickness) to parent future litters, but will spay or neuter dogs that exhibit poor characteristics or are defective. In this way, the population of dogs evolves towards a goal of having desirable characteristics.

A genetic algorithm operates in the same way, except that the individuals being bred are not dogs, but objective vectors. The computer program breeds these individual objective vectors to produce new individuals (i.e., new points in the objective space) in hope of moving the population towards a desired optimum.

The basic procedure is this: the computer creates a *population* of random *individuals*. The computer then evaluates the *fitness* of each individual in the population. Fitness is the objective function of the optimization: it is a measure of how desirable the individual's characteristics are. We want to maximize fitness. So the computer selects the fittest individuals from the population and applies genetic reproductive operations (of which the two most common are called crossover and mutation) to them to produce a new population of individuals: the second generation. The second generation is evaluated and bred again to produce the third generation, and so on. The process continues until it meets a termination criterion: perhaps it has produced a perfect individual, or has reached a maximum number of generations.

The two most common genetic operations are crossover and mutation. *Crossover* takes two individuals and exchanges some of their variables to produce two children. It is common to choose a single point in the vector, and have the two vectors exchange all variables with a higher index. This is similar to recombination in biology, where pairs chromosomes swap strands of DNA. Fig. 1(a) illustrates crossover.

Mutation takes a single individual and randomly changes one or more variables in it. It is a mechanism

*Some people, including Koza[1], compare genetic algorithms to natural selection (evolution in nature) as opposed to breeding. Normally, this is a poor metaphor, because natural selection, unlike breeding and genetic algorithms, has no defined goal. Some odd variants of genetic algorithms, however, do mimic natural selection: the individuals are given the ability to reproduce themselves and are put into an environment where hopefully individuals with the desired characteristics will survive.

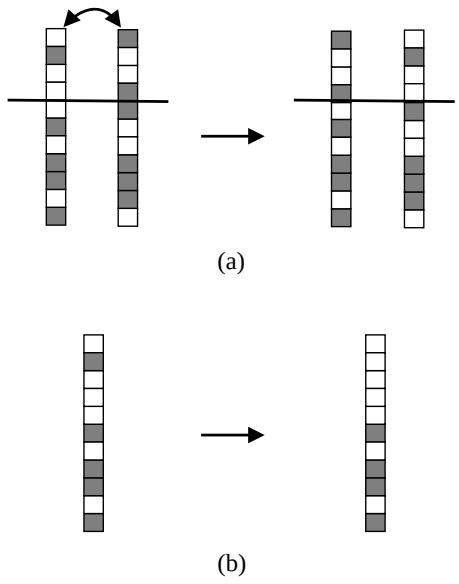


Fig. 1 Crossover (a) and Mutation (b) Operations

to ensure that certain values do not become extinct from a population. (In a nonlinear objective function, a certain value of a certain variable may not be helpful at first, and could die out; but it might be important near the optimum. Hence the need to restore it with mutation.) Fig. 1(b) illustrates mutation.

Genetic algorithms perform poorly compared to most numerical algorithms: they are more expensive computationally, and may not converge to an optimum. For purely numerical problems, it is almost always better to use some other method.

However, unlike numerical algorithms, the objective space of a genetic algorithm can span things other than numbers. Anything that can be represented digitally can be part of the objective space of a genetic algorithm: bits, integers, discrete numbers, strings of text, web pages, images, current stock prices, configuration layouts, baklava recipes, Shakespearean quotations, anagram generator programs, etc. This makes genetic algorithms very suitable for problems such as configuration optimization problems, where the different configurations cannot easily be captured in a single real number.

It also means genetic algorithms are useful when the objective space is not an orthogonal vector space, but a space of objects spanned by hierarchal tree structures.

Genetic Programming

Genetic programming is a variation of genetic algorithms where the individuals are tree-shaped structures (*trees* for short) instead of flat vectors. Genetic programming is so-called because, in early demonstrations of the technique[1], the trees represented simple computer programs*. However, trees can represent

*Any computer program can be represented hierarchally. In fact, modern compilers translate the source code into a tree structure before operating on it, a process called parsing. The tree structures in genetic programming are often called parse trees, because, although they are not formed by parsing, they look like the parse trees used by compilers.

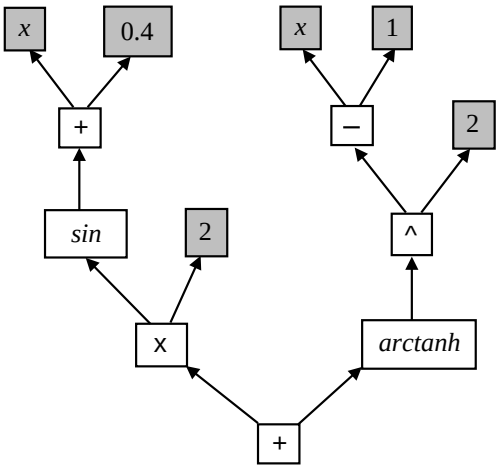


Fig. 2 Diagram of a tree representing Expression 1

any hierarchal data; they do not have to represent a computer program.

Because mathematical expressions are hierarchal in nature, a tree is ideal for storing arbitrary single-valued mathematical expressions, For example, consider Expression 1:

$$2\sin(x+0.4)+\operatorname{arctanh}(x-1)^2 \qquad (1)$$

Fig. 2 depicts this expression graphed as a tree. Each box in the figure is called a node. The shaded nodes in Fig. 2, from which spring no subtrees, are called *terminal nodes*, or leaves; they can be thought of as functions with no arguments. The unshaded nodes, from which subtrees do spring, are called *nonterminal nodes*, or branches; they represent functions of one or more arguments. Graphed this way, one can visualize how the computer performs genetic operations on trees.

Fig. 3 diagrams a crossover operation. Crossover operates on two parent trees and results in two child trees. In crossover, a node is randomly chosen in each parent tree; this node is the crossover point. Then, the parents swap subtrees at the crossover points, forming two new trees. In Fig. 3, the subtrees to be crossed are shaded. The figure depicts Expression 1 crossing with $2e^{x-4}$, where the subtrees $\operatorname{arctanh}(x-1)^2$ and $x-4$ are exchanged to yield $2\sin(x+0.4)+x-4$ and $2e^{\operatorname{arctanh}(x-1)^2}$.

Fig. 4 diagrams a mutation operation. Mutation operates on only one tree. As in crossover, a random node (called the mutation point) is selected. Then, a completely random subtree replaces the subtree at the mutation point. Figure 4 shows the subtree $4/5x$ replacinh $\sin(x+0.4)$ in Expression 1, yielding $2(4/5x)+\operatorname{arctanh}(x-1)^2$.

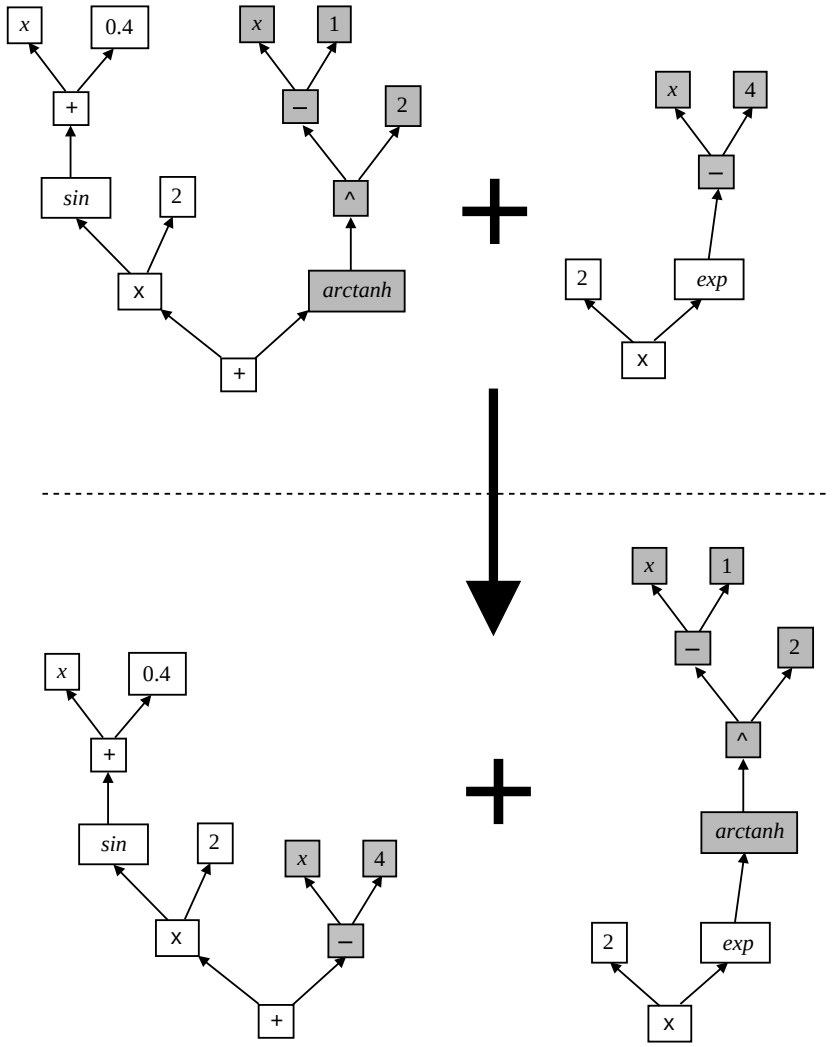


Fig. 3 Diagram of a crossover operation.

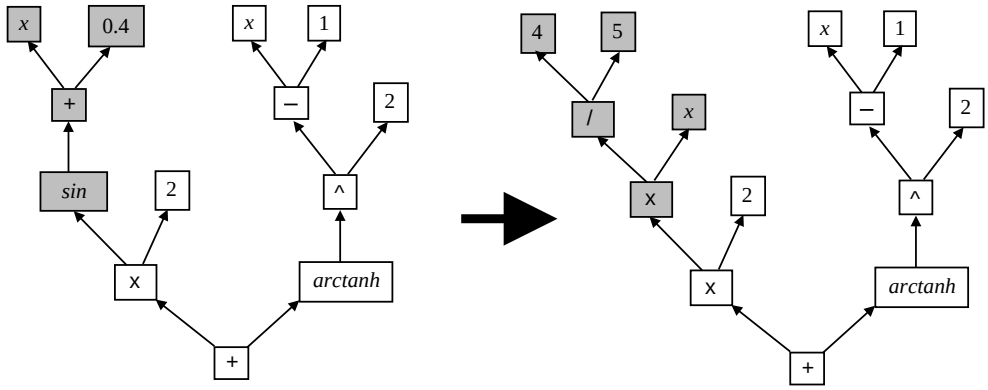


Fig. 4 Diagram of a mutation operation.

Fitness

We want to apply the technique of genetic programming to finding Lyapunov functions, where the tree structures represent potential Lyapunov functions. The population will evolve (we hope) until a Lyapunov function is produced (assuming the system is stable). But like most optimization methods, the hard part of genetic programming is not the genetic programming itself, but the evaluation of the objective function, i.e., the fitness.

The fitness of a potential Lyapunov function is a measure of its "Lyapunovness," that is, how well the function satisfies the criteria for a Lyapunov function. A continuously differentiable, real-valued function $V(x)$, where $x \in \Re^n$, is is a Lyapunov function of a system $\dot{x} = f(x)$ that is stable at the origin if the following criteria hold for some domain D containing the origin:

1. $V(0) = 0$
2. $V(x) > 0 \quad \forall\, x \in D - \{0\}$
3. $\dot{V}(x) \leq 0 \quad \forall\, x \in D$

Should $V(0)$ be defined but not equal zero, it is easy enough to define a function $\hat{V}(x) = V(x) - V(0)$ that does satisfy the first criterion. (If $V(0)$ is not defined, then $V(x)$ is not a Lyapunov function and not likely to be close. The fitness of such a function is zero.) Henceforth we assume, when evaluating the other two criteria, that $V(x)$ has been corrected by subtracting $V(0)$.

Criteria 2 and 3 are applicable to the entire domain. A function may satisfy one or both of these criteria in some places, but not others. This suggests a possible measure of fitness: the percentage of the domain D where these criteria hold. More specifically, the fitness could be the percentage of the domain satisfying Criterion 2 plus the percentage of the domain satisfying Criterion 3.

Unfortunately, determining this measure of fitness exactly is not easy for a computer to do. The most obvious way to approximate the fitness is numerically: evaluate $V(x)$ and $\dot{V}(x)$ at many points, and take the ratio of points satisfying the criteria as an approximation to the fitness. The many points could be a grid of points covering the domain D. Or, the points could be randomly chosen.

For this paper, I have implemented this fitness approximation using random points.

Lyapunov Search using Genetic Programming

And so, having examined the genetic programming, and how to measure fitness, we formulate an algorithm to search for a Lyapunov function. I have implemented this algorithm in Mathematica.

The input to the genetic programming procedure is the following: the equations of motion of the system, $f(x)$, the list of state variables $x_1, x_2, \ldots$, the domain of interest D (for my implementation, D is given by limits on the state variables), a list of mathematical functions that will be used to construct the initial random population (called the nonterminal set), and certain genetic programming parameters. The genetic programming parameters include the population size, the maximum number of generations to run, the number of test points, and other minor parameters.

The first task is to generate a random population. This requires two sets of buildings blocks: the terminal and nonterminal sets. The terminal set is the set of terminal nodes that can be used in a random function. In my implementation, this set includes the state variables x_1 through x_n, and the integers ranging from -10 to 10. I have found it very beneficial to also include the components of the right-hand-side of the equations of motion: $f_1(x)$ through $f_n(x)$. (Although the equations of motion can be complex expressions, the components always remain a single node in the tree; crossover cannot break them up.) Thus, the terminal set for my implementation is:

$$\begin{aligned} T \quad = \quad & \{x_1, x_2, \ldots, x_n, f_1(x), f_2(x), \ldots, f_n(x), \\ & -10, -9, \ldots, 9, 10\} \end{aligned}$$

Although it might be beneficial to be able to adjust this set, in my implementation this is not currently possible.

The nonterminal set is the list of function input by the user. The user should almost always include the four arithmetic operations: addition, subtraction, multiplication, division. The square function $(\cdot^2)$ is often useful because of the requirement that the Lyapunov function be positive definite. Depending on the function, other operations might be included. For example, for the damped pendulum, the sine and cosine functions might be useful because the sine appears in the equations of motion. The damped pendulum nonterminal set might look like this:

$$N = \{+, -, \times, \div, \cdot^2, \sin, \cos\}$$

There are limitless possibilities in choosing the nonterminal set, and deciding which functions to include is an art. Too many unnecessary functions decreases performance; however, not including a really useful one is crippling.

Using these two sets, the implementation generates trees recursively: the Mathematica function creating the random tree chooses a node from one of the sets randomly, and if the node is nonterminal, calls itself to generate subtrees for each of the arguments. To generate trees of manageable size (for it is easy to see that this can create very large functions), the chance of selecting a nonterminal node decreases exponentially with the depth of the recursion.

Then, having generated the population, the implementation measures the fitness of each individual. First, it evaluates each individual with all states set to zero. The result, if there is one, is to be subtracted from the individual when it is tested for positive definiteness. Individuals for which this result is undefined have their fitness set to zero.

The implementation then approximates the fitness of the rest of the individuals. In both cases, it tests a random set of points for satisfaction of the two criteria $V(x) > 0$ and $\dot{V}(x) = (\partial V/\partial x)^T f(x) \leq 0$. (Incidentally, because Mathematica can differentiate symbolically, it can evaluate $\dot{V}(x)$ without using finite differences.) The approximate fitness is the ratio of points satisfying $V(x) > 0$ plus the ratio of points satisfying $\dot{V}(x) \leq 0$, divided by two.

The number of points tested depends on how well the points satisfy the criteria. The implementation first tests a set of random points for one criterion. If this ratio is less than 1/2, then the ratio is accepted as the approximation. Otherwise, it tests another set of points. If less than 3/4 of the points in both sets satisfy the criterion, then that ratio it is accepted as the approximation. Otherwise a third set is tested. If less than 7/8 of all the points satisfy the criterion, that ratio is accepted at the approximation. Otherwise a fourth set is tested, and so on. This method frees the computer from wasting time on the least fit individuals, while giving precise results for the most fit individuals. The number of random points per set, and maximum number of sets, are settable parameters.

Having measured each individual's fitness, the genetic programming procedure breeds a new population. To breed a fitter population, the procedure tends to choose individuals with higher fitness, but also allows some less-than-optimal individuals to breed to retain genetic diversity. My implementation uses tournament selection, where it randomly selects some number (a settable parameter) of individuals. Of those selected, the individual with the highest fitness gets to breed.

The size of the new population is the same as the size of the original one*. The implementation creates most of the individuals in the new population by crossing two parents, both of whom have been selected by tournament. A small percentage of new individuals undergo mutation after crossover. The mutation rate is usually very small, less than two percent, and is a settable parameter.

Some individuals in the new population are not formed by crossover, but are instead a copy of a single parent (but still subject to mutation). The purpose of such asexual reproduction is to save the population from serious regression in the case of an unluckily bred generation. A small percentage of individuals in the new population is formed this way. The percentage should be small number, around ten percent, and it is a settable parameter.

After the generation of the new population, the algorithm loops back to measuring its fitness. It continues looping, and hopefully finding better individuals with each generation, until some termination criterion is met. My implementation has two termination criteria. One is when a maximum number of generations, specified by the user, has been run. The other is the appearance of an individual with a perfect score: all of the points tested satisfy the Lyapunov criteria. The procedure stops and returns the fittest individual when termination occurs.

Because genetic programming is a heuristic method, there is no guarantee that it will find a true Lyapunov function. This problem is deepened by the fitness measure not being completely accurate. We hope that the problem is not serious.

Testing and Results

I applied my genetic programming implementation to three different nonlinear systems: a simple polynomial model, a damped pendulum, and a nonlinear system containing the tangent function. All three were two-dimensional, so that the returned function could be plotted.

The parameter settings used in the tests were as follows:

- The generation size varied. It was usually 500; but for some runs was 1000, others 200.
- The maximum number of generations was 20.
- The number of random points in a test set was 20. The maximum number of sets that could be tested was 10, making the maximum number of random points tested 200.
- The number of individuals selected for the tournament was usually 4, although it was 2 in a few runs.
- 90 percent of crossovers happen at a nonterminal node.
- The mutation rate was 2 percent.
- For each generation, 10 percent of the individuals are direct copies of one parent in the previous generation; the other 90 percent are formed by crossover.

The results, in general, were good, if uncertain. There were more false positives that I would have liked. In most cases, convergence was faster than I expected.

*I want to try an initial population somewhat larger than the regular population size. This is because most of the initial population have a fitness of zero, and thus the first generation is mostly the offspring of a small number of individuals. A larger initial population could increase the genetic diversity.

The functions produced were sometimes too complicated to analyse; therefore, many of the results are graphed. Figures 5–10 show graphs of some of the functions produced by the tests. Each figure shows four graphs. In each figure, the graph on the top-left plots the returned function $V(x)$, which is hopefully a Lyapunov function. Next to it, the graph on the top-right plots $\min(V(x), 0)$; this plot makes it easy to see where $V(x) < 0$. The graph on the bottom-left plots $\dot{V}(x)$. Next to it, the graph on the bottom-right plots $\max(\dot{V}(x), 0)$, to make it easy to see where $\dot{V}(x) > 0$. We want both graphs on the right to be flat.

Needless to say, because of the granularity of the plots, it is possible that there are regions that cross the plane but are missed by the plotting, especially near zero. In fact, I noted such missed details a few times when examining the returned function for simple systems. For most systems, absolute proof of these functions' Lyapunovness requires painstaking algebraic or numerical analysis.

Simple Polynomial Model

The first system tested was the nonlinear system given by Eqs. 2 and 3 (which had appeared in a homework problem):

$$\dot{z}_1 = z_1 z_2 \quad (2)$$
$$\dot{z}_2 = -z_1^2 - z_2 \quad (3)$$

The ranges of z_1 and z_2 are $z_1 \in [-1, 1]$ and $z_2 \in [-1, 1]$. For two of the runs, I used the nonterminal set $\{+, -, \times, \div\}$; for the rest, $\{+, -, \times, \div, \cdot^2, \sqrt{\cdot}\}$.

One test run returned the function $V(z_1, z_2) = z_1^2 + z_2^2$, which is indeed a Lyapunov function in the domain. It is clearly positive definite, and its derivative is

$$\begin{aligned} \dot{V} &= 2z_1^2 z_2 + 2(-z_1^2 - z_2)z_2 \\ &= -2z_2^2 \leq 0. \end{aligned}$$

However, it only proves stability, not asymptotic stability, because $\dot{V} = 0$ on the z_1-axis. Another test run returned the function $V(z_1, z_2) = (z_1^2 + z_2^2)^2$, which is also a Lyapunov function, but again does not prove asymptotic stability. Fig. 5 shows the plots of this Lyapunov function.

There were a couple runs that produced functions that looked Lyapunov from the graphs, and seemed have $\dot{V}$ strictly less than zero except at the origin, which would prove asymptotic stability. They were somewhat more complex to analyse. Fig. 6 shows the plots for one of these functions. (However, I suspect this function has a small region where it fails to satisfy the Lyapunov criteria near the origin.)

But not all of the tests returned a Lyapunov-looking function. One test produced a function that (seemed as if it) met the three Lyapunov criteria; however, it wasn't continuously differentiable (see Fig. 7). On some other runs, the system didn't converge, but reached the maximum number of generations while only satisfying the Lyapunov criteria on 80% or so of the domain. The two runs I made without the square function in the nonterminal set did this. This is the nature of genetic algorithms, because of their randomness: getting a good result often requires several runs.

Damped Pendulum

Next, genetic programming tackled the damped pendulum system, given by Eqs. 4 and 5:

$$\dot{x}_1 = x_2 \quad (4)$$
$$\dot{x}_2 = -\sin x_1 - x_2 \quad (5)$$

The ranges for x_1 and x_2 are $x_1 \in [-\pi/2, \pi/2]$ and $x_2 \in [-1, 1]$. The nonterminal set was $\{+, -, \times, \div, \cdot^2, \sqrt{\cdot}, \sin, \cos, \arctan\}$.

For this system, the tests produced direct hits quite quickly (not more than eight generations) in almost all cases. Figs. 8 and Fig. 9 show the results of two test runs.

Fig. 8 shows one of the oddest functions produced: a function composed of a lot of arctangents. This the interesting thing about genetic programming: it can produce functions that don't look at all as we would expect them. But, assuming this function did not latently stray across the $V = 0$ plane, it proves the stability nonetheless.

Fig. 9 shows a false hit. This function was returned as 100% Lyapunov, although there are regions of positive $\dot{V}$ near the x_2-axis, which weren't tested and slipped through. The fitness approximation needs improvement.

Nonlinear system containing the tangent function.

Finally, the system given in Eqs. 6 and 7 was tested:

$$\dot{x}_1 = -\tan x_1 + x_2^2 \quad (6)$$
$$\dot{x}_2 = -x_2 + x_1 \quad (7)$$

The ranges on x_1 and x_2 are $x_1 \in [-1, 1]$ and $x_2 \in [-1, 1]$. The nonterminal set was $\{+, -, \times, \div, \cdot^2, \sqrt{\cdot}, \sin, \cos, \tan\}$.

This, of all the systems, had the oddest-looking results. I ran four tests, and three returned results that appeared, from the graphs, to satisfy the Lyapunov criteria. The results of the three good test runs were:

$$\sin\sin\tan\sin(x_1 + x_2^2 - \tan x_1) + (x_2^2 - \tan x_1)^2$$
$$0.9955 + \cos\left((\cos x_1 + \sin\sin\cos(x_2^2))^2\right)$$
$$1.557 + (x_2^2 - \tan x_1)^2 - \tan\cos\tan x_2$$

Fig. 10 shows the first result listed above.

The unusual form of these results suggest that genetic programming is useful for expanding the range of possibilities when searching for Lyapunov functions. A human trying to find a Lyapunov function analytically would tend not to use oddities like $\sin\sin\tan\sin(x)$; yet genetic programming can search these obscure recesses of hierarchal function space.

Conclusion and Future Work

It is clear that, even with an imperfect fitness measure, genetic programming is capable of finding functions that at least satisfy the Lyapunov properties over most of their domain. I expect that the method, with a better measure of fitness, will be able to find true Lyapunov functions without false hits. How well this method will scale up to higher-dimensional and more complex system remains to be seen.

Some possible future work, besides testing it on more complex systems, is:

- The measure of fitness has to improve. Perhaps there is a way to replace the approximation with an exact measure. One possibility is interval arithmetic, a feature of Mathematica. Although it produces intervals too conservative (a true Lyapunov function might return a interval with negative values), it warrants more study.

- Sometimes a Lyapunov function with vastly disparate scale would be produced. In some directions it would rise to 10^{20}; in others it would be only in the tens. It could be a problem when the Lyapunov function is used in the control law: in one direction, the slightest error could saturate the controls; in another direction, the control law could be almost completely numb to large error.

  As a remedy, the Lyapunov function could be subject to more stringent conditions, such as being bound above and below by some positive definite function.

- Incorporating genetic programming into robust nonlinear control design. Genetic programming could search for a control law at the same time as Lyapunov functions. This would require a more advanced genetic programming structure, where a single individual contains multiple trees: one for the Lyapunov function, and others for the controls.

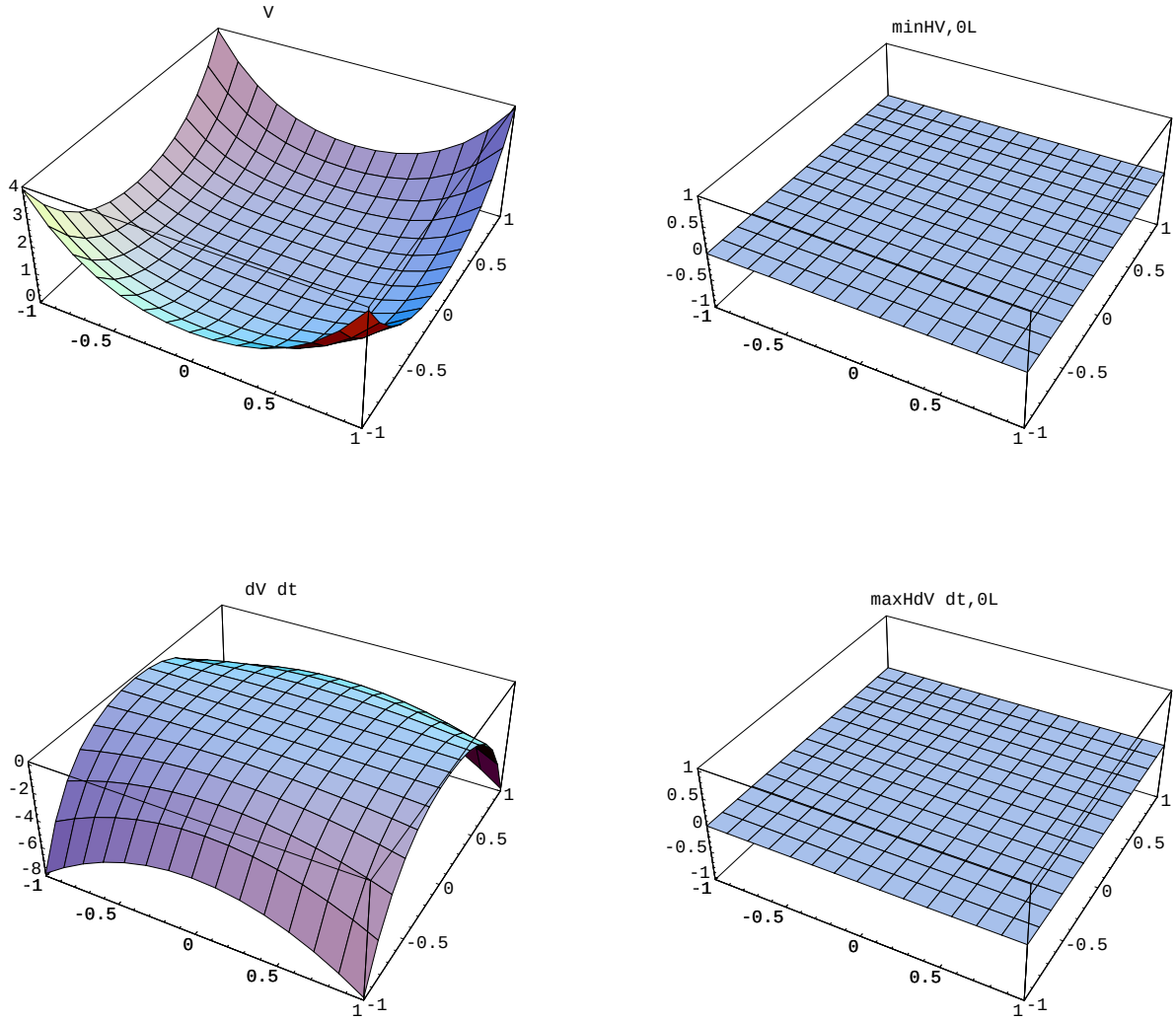
References

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

2. Rodriguez-Vazquez, K. and Fleming, P. J. "Multi-objective genetic programming for nonlinear system identification," *Electronics Letters,* Vol. 34, No. 9, 1998, pp. 930–931.

3. Imae, J. and Takahashi, J. "Design method for nonlinear H_∞ control systems via Hamilton-Jacobi-Issacs equations: a genetic programming approach," *Proceedings of the IEEE Conference on Decision and Control,* Vol. 4, IEEE, Piscataway, New Jersey, pp. 3782–3783.

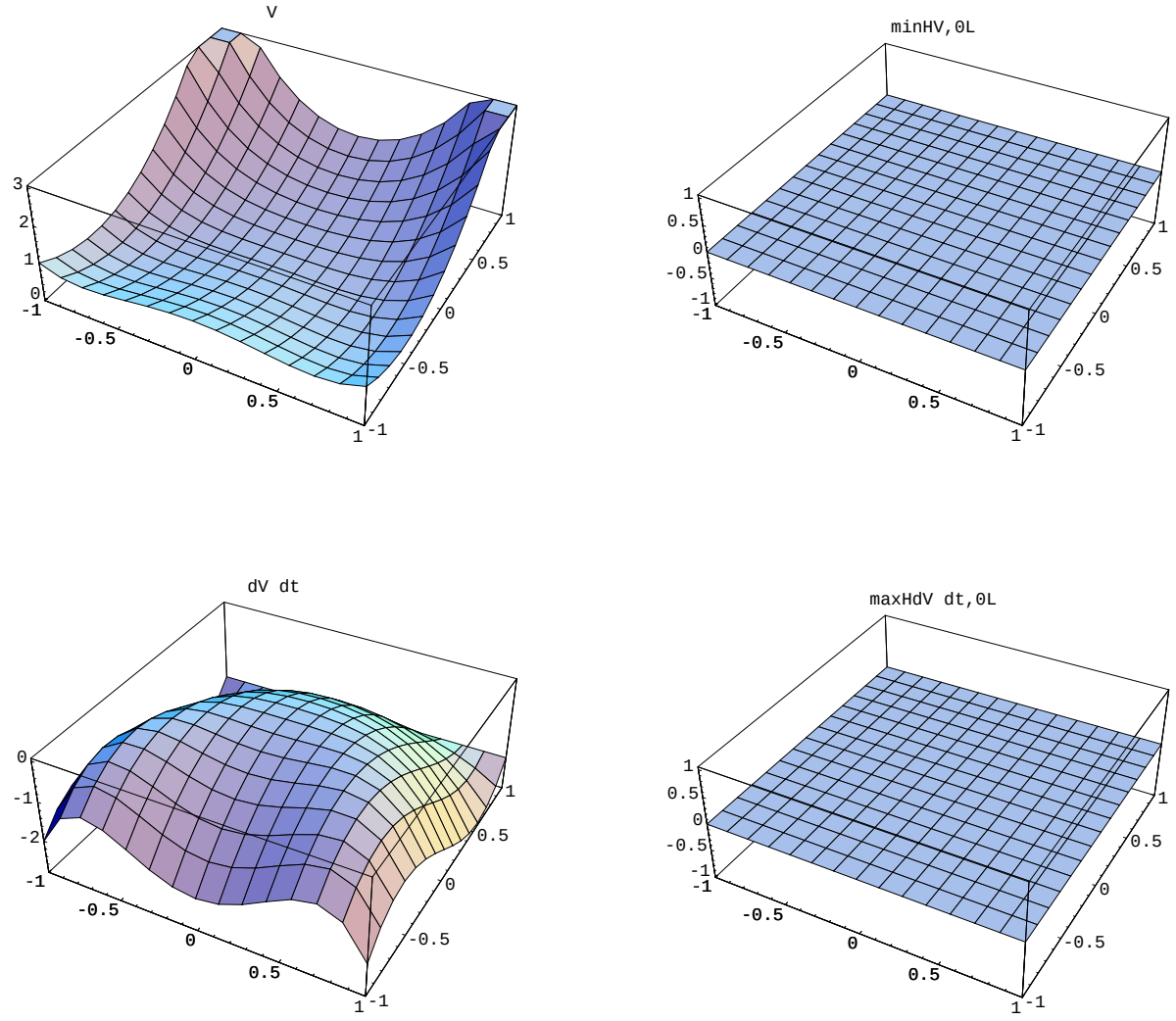
4. Khalil, H. S. *Nonlinear Systems,* 2nd Ed., Prentice Hall, Upper Saddle River, New Jersey, 1996, p. 100.



$$V(z_1, z_2) = (z_1^2 + z_2^2)^2$$

$$\begin{aligned} \dot{z}_1 &= z_1 z_2 \\ \dot{z}_2 &= -z_1^2 - z_2 \end{aligned}$$

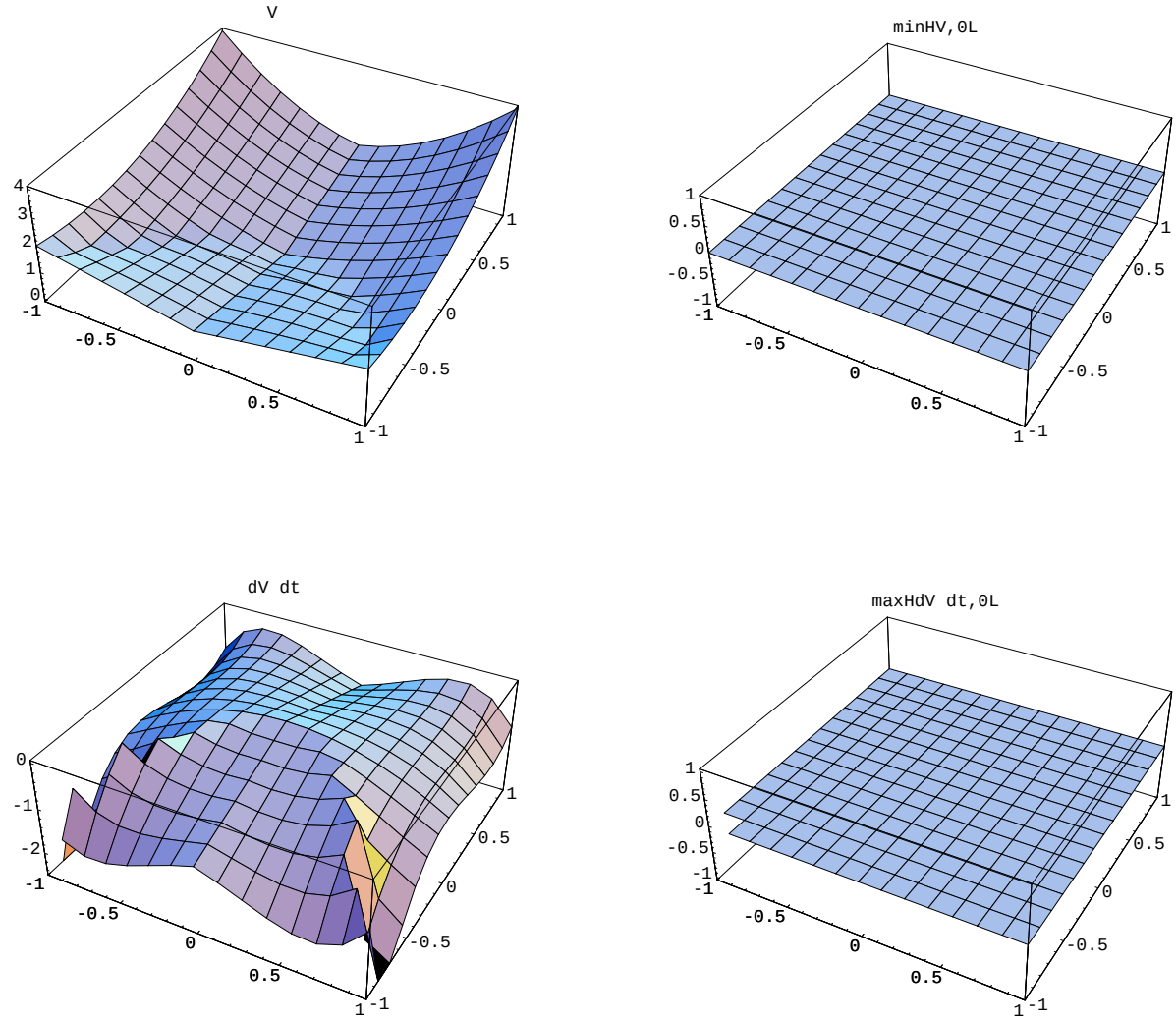
Fig. 5 Result of a genetic programming Lyapunov search for a simple polynomial system.



$$V(z_1, z_2) = (-z_1^2 - z_2)^2 + z_1^2 z_2^2$$

$$\begin{aligned} \dot{z}_1 &= z_1 z_2 \\ \dot{z}_2 &= -z_1^2 - z_2 \end{aligned}$$

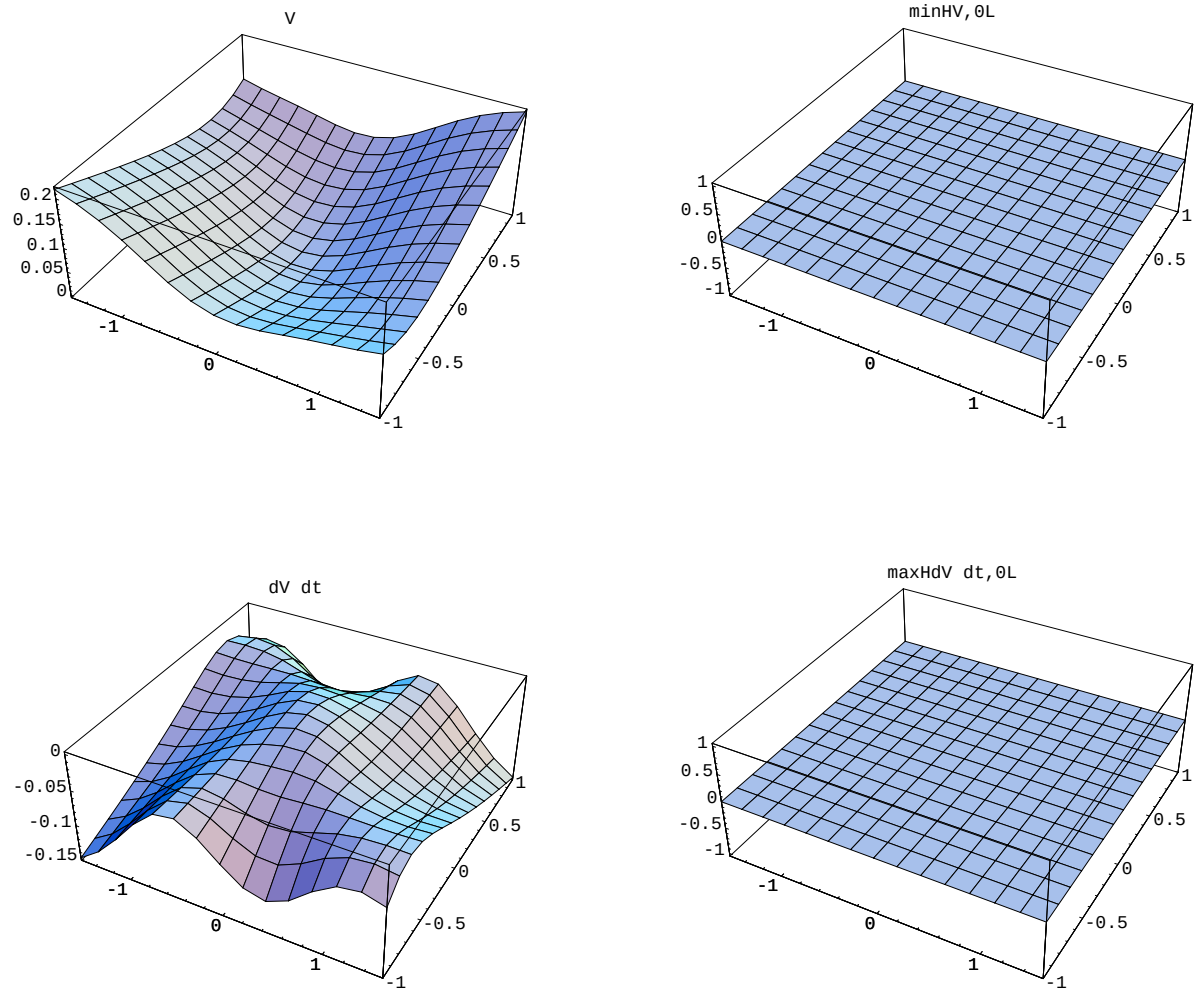
Fig. 6 Result of a genetic programming Lyapunov search for a simple polynomial system.



$$V(z_1, z_2) = \sqrt{z_1^2} + \sqrt{(-z_1^2 - z_2)^2} + z_1^2 z_2^2$$

$$\begin{aligned} \dot{z}_1 &= z_1 z_2 \\ \dot{z}_2 &= -z_1^2 - z_2 \end{aligned}$$

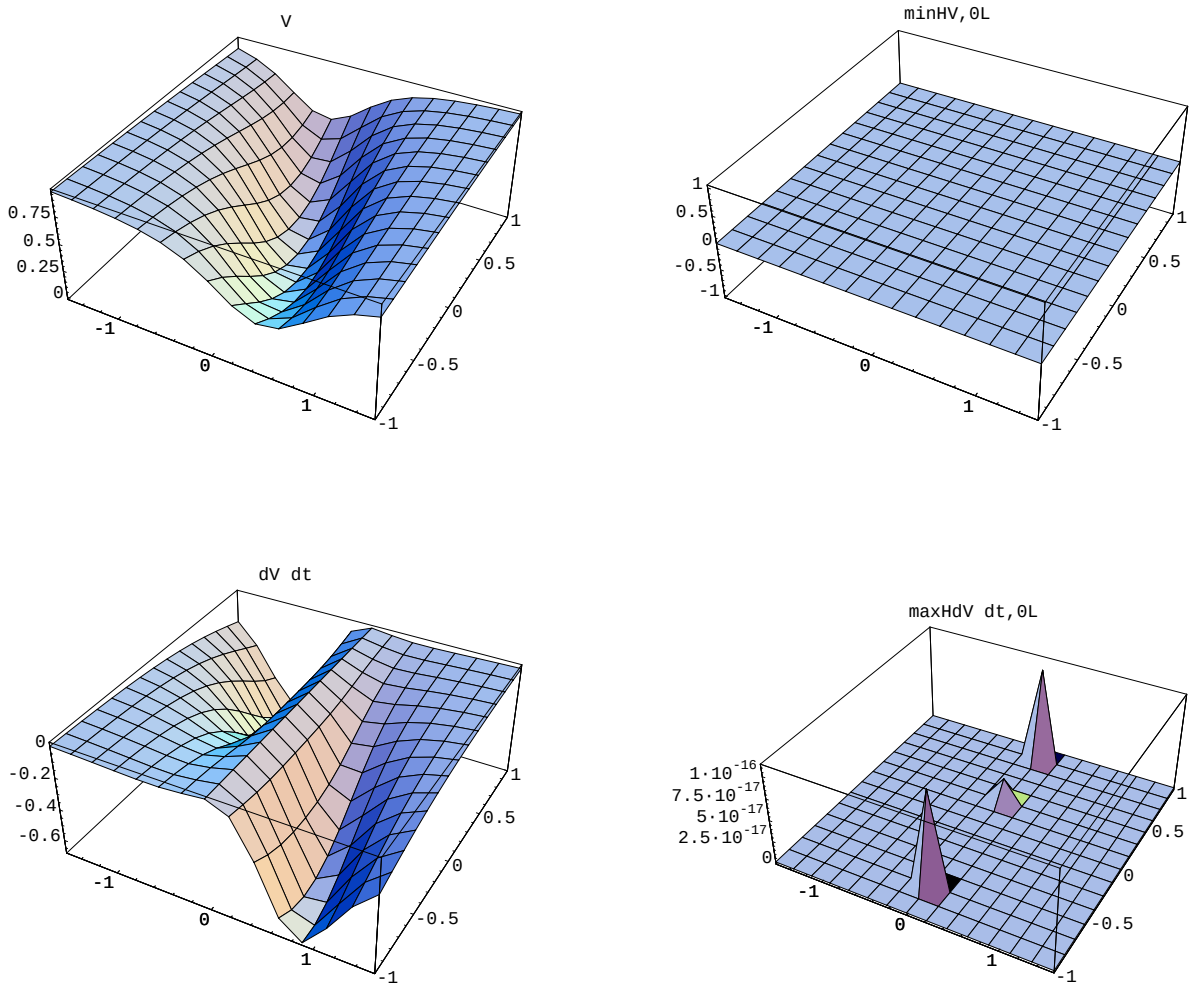
Fig. 7 Result of a genetic programming Lyapunov search for a simple polynomial system.



$$V(x_1, x_2) = \left(\arctan(x_1 \arctan x_1) + \frac{x_2 \arctan(x_2 + \sin x_1)}{\sqrt{1 + (\arctan(x_2 + \sin x_1))^2}} \right) /8$$

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

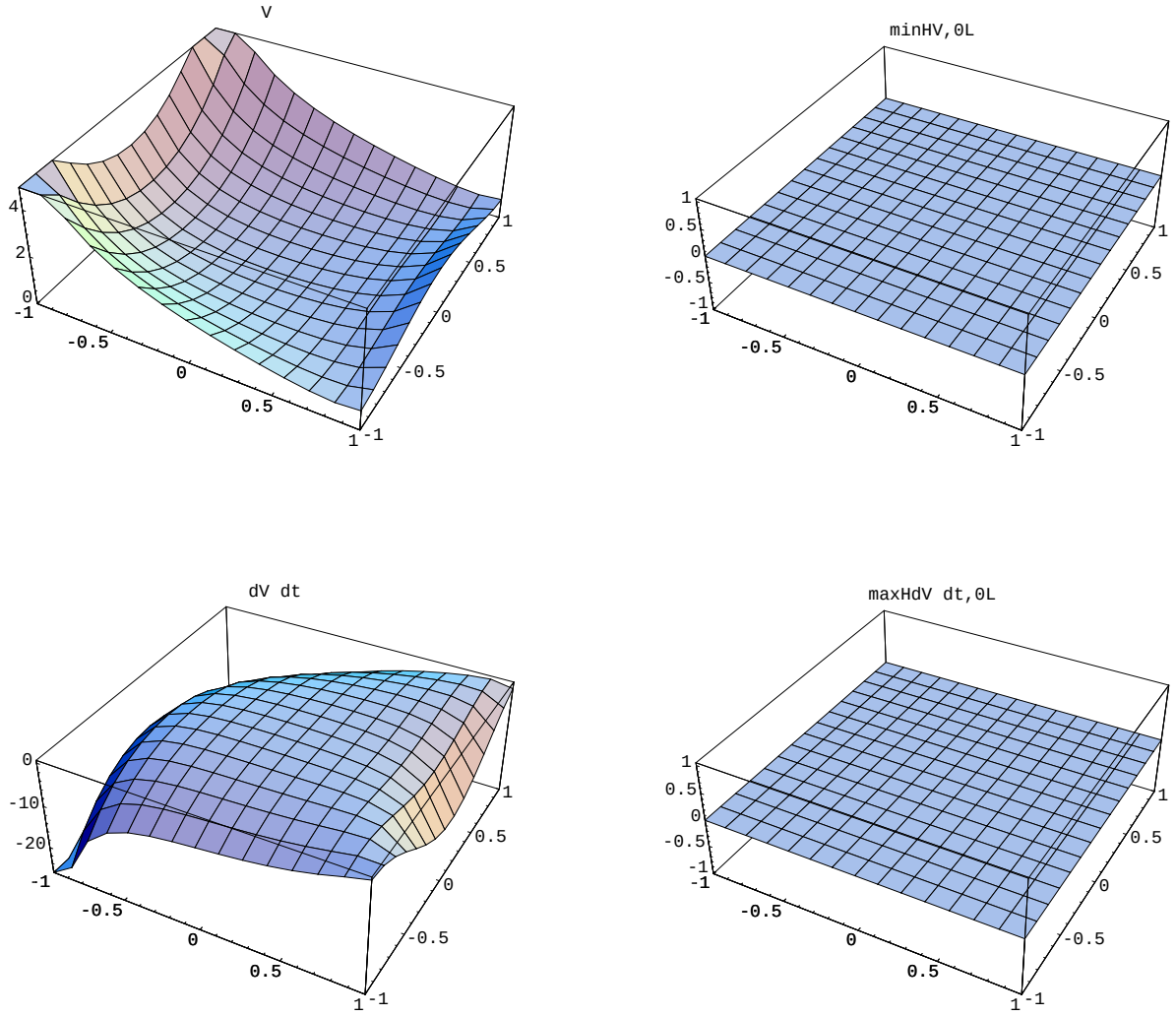
Fig. 8 Result of a genetic programming Lyapunov search for a damped pendulum system.



$$V(x_1, x_2) = \arctan(\arctan(x_1^2 + (x_1 + x_2)^2))$$

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

Fig. 9 False result of a genetic programming Lyapunov search for a damped pendulum system.



$$V(x_1, x_2) = \sin\sin\tan\sin(x_1 + x_2^2 - \tan x_1) + (x_2^2 - \tan x_1)^2$$

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

Fig. 10 Result of a genetic programming Lyapunov search for a nonlinear system containing the tangent function.