Deep Lyapunov Functions and Symbolic Regression

Sungyoon Kim

Department of Electrical and Computer Engineering

Seoul National University

Abstract

Using machines to assist mathematicians in their research has met a new phase with the huge success of deep learning. One can think of applying deep learning to find a Lyapunov function for the given equilibrium of a dynamics, but current deep learning methods cannot help to find an analytic form of the Lyapunov function, thus failing in helping mathematicians actually prove stability. In this paper, we apply advanced symbolic regression techniques on the learned Lyapunov function to find the analytic form of it. The method can be widely used for complicated, high dimensional dynamics, thus can serve as a useful tool when the Lyapunov function of the given dynamics is unknown.

1 Introduction

The tremendous success of deep learning led mathematicians to use the technology on solving problems that seemed impossible in the past. Solving PDEs with neural networks became a newly rising field([1],[2],[3]), and there has been pioneering works to use deep learning models to find the hidden invariance behind a dynamical system([4]), or to find counterexamples of a combinatorics problem ([5]). It is no lie to claim that we are living in a world where machines can work as a decent assistant to mathematicians.

For mathematicians that study dynamics, understanding the stability of the equilibrium of a given dynamical system is an important problem. One way to prove that an equilibrium is stable is the so-called 'direct Lyapunov method', where a mathematician should carefully design a Lyapunov function V that satisfies

$$V(X_e) = 0, V(X) > 0, DV(X) < 0 \quad or \quad DV(X) \leq 0 \quad \forall X \in N - X_e.$$

Here, X_e is a given equilibrium, and N is a certain neighborhood of X_e. When $DV(X) < 0$ holds, X_e becomes the asymptotic stable equilibrium, and when $DV(X) \leq 0$ holds, X_e becomes the Lyapunov stable equilibrium. For some dynamics the Lyapunov function is intuitive, but for others finding it can be a very difficult problem.

It may not be a surprise that there are works to automatically approximate Lyapunov functions with deep learning. In fact, there have been numerous works that finds Lyapunov function of an equilibrium with deep learning, and use the learned function for stable control. However, one intrinsic limitation of deep Lyapunov methods is that we cannot strictly prove whether a given equilibrium is stable or not. Even after sufficient amount of training, and after we obtain the resulting function $V(X;\theta^*)$ that satisfies the constraints for every training data, it may fail to meet the given conditions outside the sampled data. Unless we use very simple networks and analytically prove the given conditions, there is no way we can solidly claim that we have truely found the Lyapunov function. Though finding the Lyapunov function with deep learning may serve a prominent role engineering, its application to pure math is simply implausible.

For machines to help mathematicians prove the stability of a given equilibrium with direct Lyapunov method, they should be able to provide at least the analytic form of the obtained Lyapunov function. Genetic programming and symbolic regression can serve as promising ways to solve such problem. There has been a few works that aims to find an analytic form of the Lyapunov function by genetic programming. However, genetic programming is highly volatile and frequently fails when there are too many variables. Thus, those methods only work for very low-dimensional, simple dynamics.

In this thesis, I aim to present a novel method that can provide an analytic form of more complicated, higher dimensional dynamics. The method is rather simple: First, approximate a Lyapunov function with deep neural networks. Second, use advanced methods in symbolic regression to find the analytic form of the Lyapunov function. With the method we can fully exploit the expressionability of neural networks, and can evade the problems that genetic algorithms have by using symbolic regression algorithms that are not based on genetic programming.

The rest of the thesis is divided into three parts: In Preliminaries, I give a survey on existing deep Lyapunov methods, their application to stable control. I also give a survey on symbolic regression and how they can be applied to find an analytic form of the Lyapunov function. In Experiments, I propose a novel algorithm to find the analytic form of Lyapunov functions, and provide extensive experimental results where the algorithm successfully finds either the vanila / strict Lyapunov function. At last, in Conclusion, I will summarize the work and present possible further research topics.

2 Preliminaries

2.1 Deep Lyapunov Methods

This Section will cover different methods to approximate Lyapunov functions with neural networks and provide an application where deep Lyapunov functions are extensively used. In Section 2.1.1, I will provide the necessary backgrounds of deep learning. After that, in Section 2.1.2, I will illustrate the general idea behind deep Lyapunov methods and their intrinsic limitations. In Section 2.1.3 I will introduce some earlier approaches that proposed to estimate Lyapunov functions with neural networks, and drawbacks of these approaches. At last, in Section 2.1.4, I will elaborate on the modern studies that aim to overcome the drawbacks, along with how deep Lyapunov functions are extensively applied in the field of control and robotics.

2.1.1 Preliminaries in deep learning

One of the primary objectives of deep learning is to find a good approximation of a function $f : \mathbb{R}^d \to \mathbb{R}$ from a class of functions $\mathcal{F}$. A typical formulation of a deep learning problem is as the following:

Suppose $f : \mathbb{R}^d \to \mathbb{R}$, $\mathcal{F}$ is given as $\mathcal{F} = \{V(x;\theta)|\ \theta \in \mathbb{R}^N\}$(where θ denotes the trainable parameters), and we have sampled data $D = \{(x_i, y_i)|\ i = 1, 2, ..., n\}$ from the distribution $\mathcal{D} = \{(x, f(x))|\ x \in \mathbb{R}^d\}$. Find the optimal parameter θ^* that minimizes

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(V(x_i;\theta), y_i),$$

where l denotes the error between $V(x_i;\theta)$ and y_i. An example of l can be the distance between $V(x;\theta)$ and y, given that Y is a metric space.

$V(x;\theta)$ is given as a composition of 'multiple layers', which is the reason why we call these models 'deep neural networks'. For each layer $k = 1, 2, ..., n$, a vector x in $\mathbb{R}^{d_{k-1}}$ is given as an input. Here, d_k is called the size of the hidden unit of layer k when $k \geq 1$, and d_0 is simply the size of the input. The input passes through an affine transformation

$$A_k(x) = W_k x + b_k,$$

where $W_k \in \mathbb{R}^{d_{k-1}\times d_k}$ and $b_k \in \mathbb{R}^{d_k}$, and then passes through a nonlinear function σ at each layer. tanh, relu($max\{x, 0\}$), sigmoid are the conventionally used nonlinearities. When there are n layers, we have affine transformations A_1, A_2, ..., A_n with the corresponding parameters (W_1, b_1), (W_2, b_2), ..., (W_n, b_n). The overall architecture is given as

$$V(x;\theta) = A_n(\sigma(A_{n-1}(...(A_2(\sigma(A_1(x)))...),$$

where the trainable parameters θ are $\{W_1, b_1, W_2, b_2, ..., W_n, b_n\}$.

The training process is mostly done by optimization methods which are often modifications of gradient descent. As the size of the data is too large, we cannot apply vanila gradient descent on the loss itself. Instead, at each step of the optimization process, we sample smaller 'batches' of the data and apply gradient descent on the batch. The appropriate batch size is a hyperparameter that has different optimal values for different tasks. The learning rate, momentum, and epochs are also hyperparameters related to the training process: The Learning rate is the size of the step that the optimizer uses for gradient descent, the momentum is a parameter that momentum-based optimizers such as Adam use, and the epoch is the number of repetitions to see the whole data.

2.1.2 Deep Lyapunov Methods: A general explanation

Suppose the dynamics

$$X' = F(X).$$

is given, where $X \in \mathbb{R}^d$. Without loss of generality, let's say the dynamics attains equilibrium at $X = 0$. As briefly explained in the introduction, the objective of direct Lyapunov analysis is to

analyze the stability of a given equilibrium by finding an appropriate Lyapunov function $V : \mathbb{R}^d \to \mathbb{R}$. When there is a function V that satisfies

$$V(0) = 0, V(X) > 0, DV(X) < 0 \quad \forall X \in N_0 - \{0\}. \tag{1}$$

It can be shown that the equilibrium $X = 0$ is asymptotically stable. Here, $DV(X)$ is the directional derivative of V at X, and N_0 is a certain neighborhood of $X = 0$. Similarly, when there is V that satisfies $DV(X) \leq 0$ instead of $DV(X) < 0$, we can show that $X = 0$ is stable in the sense of Lyapunov.

Deep Lyapunov methods aim to approximate the Lyapunov function of a given equilibrium by training a neural network with data sampled from the dynamics. The idea behind deep Lyapunov methods is using a specially designed loss function $L(\theta)$ that penalizes $V(X; \theta)$ when it violates the conditions of (1). Though the design of loss functions may vary, they are essentially derived from the conditions of (1).

To illustrate further, I will give an example of a naive loss function $L(\theta)$ and how Lyapunov functions can be learned by the training process. The most naive loss function one can think of, which is referred to as the 'Lyapunov risk([6])', is given as the following:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} (V(0; \theta)^2 + \max\{0, -V(X_i; \theta)\} + \max\{0, \nabla_X V(X_i; \theta) \cdot F(X_i)\}). \tag{2}$$

Here, $\{(X_i, F(X_i))|\ i = 1, 2, ..., n\}$ are the data points sampled from the phase space. Minimizing $V(0; \theta)^2$ decreases the absolute value of $V(0; \theta)$, which makes the parameters move toward satisfying constraint (1). Similarly, to minimize $\max\{0, -V(X_i; \theta)\}$, increasing $V(X_i; \theta)$ above 0 is beneficial, which enforces the parameters to satisfy the positive definiteness of V. At last, to minimize $\max\{0, \nabla_x V(X_i; \theta) \cdot f(X_i)\} = \min\{0, DV(X_i)\}$, decreasing $DV(X_i)$ under 0 is beneficial, which enforces the parameters to satisfy the negative definiteness of DV. Thus, when we enforce θ to decrease the loss function $L(\theta)$, θ is enforced to satisfy the constraints in (1), and the resulting function $V(X; \theta^*)$ after the training process becomes a likely candidate of the Lyapunov function at equilibrium $X = 0$.

Using the naive loss function, we can train a simple three-layer neural network to obtain the Lyapunov functions of the dynamics

$$\begin{cases} x' = y, \\ y' = -x. \end{cases} \tag{3}$$

at $(x, y) = (0, 0)$. Here, I used a three-layer network with hidden units [128, 32, 8], each followed by a softplus activation function. For training, I used the ADAM optimizer with a learning rate of 1e-3, a batch size of 256, and proceeded training for 10 epochs. All the training was done using Google Colab. The value of the trained Lyapunov function $V(x, y)$ and the directional derivative $DV(x, y)$ can be seen below. The surface corresponds to $V(x, y)$, and the wireframe corresponds to $DV(x, y)$.

We can see that the learned Lyapunov function satisfies the constraints $V(X) \geq V(0)$, $V(0) \approx 0$ and $DV(X) \leq \epsilon$ where $\epsilon \approx 0.001$, though $V(0) = 0$ and $DV(X) \leq 0$ is not strictly satisfied. Even so, we can see from Figure 8 that the learned function shows properties that a Lyapunov function is expected to have.

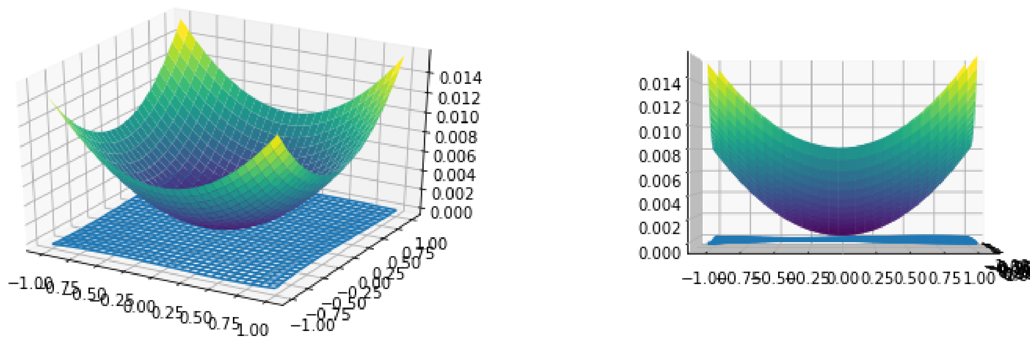


Figure 1: Learned Lyapunov function and its directional derivative

For very simple dynamics such as (3), we can approximate the Lyapunov function with a naive approach as in (2). However, for larger and more complicated systems, a more stable and scalable strategy for training is necessary, which motivates using different mathematical properties of Lyapunov functions to design the loss function.

Even though we use more sophisticated training strategies, the training process of Lyapaunov functions may fail. One possible reason is that the equilibrium $X = 0$ is not stable, thus there doesn't exist a function that satisfies the conditions of Lyapunov functions. Another possibility is that the training data is insufficient and the training process has not converged properly. At last, the model structure or hyperparameters may have been unsuitable. It is the job of the engineer who trains the model to decide what the issue is, and make decisions on the stability of the equilibrium.

2.1.3 Some Earlier approaches

The earliest approaches in the literature give theoretical possibilities of neural networks approximating Lyapunov functions([7],[8],[9]). The motivation is straightforward, as neural networks are universal function approximators([10]) and can approximate any function given sufficient number of nodes. [7] is the earliest paper to use neural networks to understand a given dynamics. In [7], the author suggests a specially designed neural network of which he calls the 'Lyapunov Machine', and discusses how the machine can solve problems in nonlinear dynamics, such as testing global asymptotic stability and isolating domains of local asymptotic stability. The author also demonstrates that these Lyapunov Machines can be trained to approximate Lyapunov functions, yet not showing any empirical results. Although [7] is the first to use neural networks to find Lyapunov functions, the method is distinct to the typical approach when we use neural networks to approximate functions.

The approach that we typically use was first introduced in [8]. In [8], the authors introduce the concept of multi-linear perceptrons(or MLPs for short), and show the possibility of MLPs approximating Lyapunov functions. Another approach in [9] suggests recursive neural networks as an approximator for Lyapunov functions. The algorithm in [9] is as follows: First, sample a discrete trajectory $x(t_k)$, and calculate the value $V(t_k)$ at time t_k. If $V(t_{k+1}) < V(t_k)$, do nothing. If not, adjust the weights of neural networks to decrease the value of $V(t_{k+1}) - V(t_k)$. Repeat until $V(t_{k+1}) < V(t_k)$ holds for sufficiently large number of starting points. The algorithm exploits the fact that $DV(X) < 0$ should

hold by enforcing $V(t_{k+1}) < V(t_k)$ for the sampled trajectories(thus, the value of the Lyapunov function should strictly decrease with the trajectory), and also proposes to use recurrent neural networks based on the fact that the trajectory of the dynamic system is also recursive.

After the earliest research that proposes the possibility of neural networks as approximators of Lyapunov functions, there have been several works that uses neural networks to train very simple neural networks and find appropriate Lyapunov functions of a given equilibrium([11],[12]). In [11], the authors use a sufficient condition for Lyapunov functions to train a neural network with a single hidden layer. Specifically, we know that when there exist function $V(X) : \mathbb{R}^d \to \mathbb{R}$ that satisfies

$$V(0) = 0, \nabla_X V(0) = 0, \nabla_X^2 V(0) \succ 0, -\nabla_X^2 DV(X) \succ 0. \tag{4}$$

then the function $V(X)$ becomes the Lyapunov function of the given dynamics $X' = F(X)$. The property can easily be proved by the second derivative test on $V(X)$ and $DV(X)$. The authors of [11] find analytic formulae for $V(0)$, $\nabla_X V(0)$, $\nabla_X^2 V(0)$, and $\nabla_X^2 DV(X)$ to find the constraints given to the parameters of the neural network. Finding the analytic form is possible as they use a very simple network with a single hidden layer. To enforce positive definiteness of the Hessian matrices, they use a fitness function that has greater value when the Hessian is positive definite. They suggest two examples bases on the eigenvalues λ_i of the Hessian: One is $\sum_{i=1}^{n} e^{\lambda_i}$, and another is $\sum_{i=1}^{n} \min\{\lambda_i, 0\}$. After that, they use genetic algorithms as an optimizer to increase the fitness function, subject to the constraints in (4). They also provide numerical experiments of their purposed method, and shows that their method can help find a Lyapunov function for various systems.

[12] uses approximation theory to first choose a basis of approximators, and use neural networks to find the weights of the linear combination of the basis elements. From Stone-Weierstrass theorem, we know that the family of polynomials serve as a good approximator for continuous functions. Using the fact, the authors fix the neural network as a polynomial with unknown coefficients, and train the neural network to find only the coefficients. To train the network, [12] uses the fact that the Lyapunov function should decrease with the positive trajectory, similarly to the approach in [9]. The algorithm they suggest is as below:

Algorithm 1 Polynomial regression of Lyapunov functions([12])

1: **for** $p = 0, 1, 2, ...$ **do**
2:     Obtain trajectories $X_p(t_k)$ for $k = 0, 1, ..., N$
3:     $X_p(t_N) \approx 0$ should hold
4: **end for**
5: **for** $k = 0, 1, 2, ..., N$ **do**
6:     Assign $V(X_p(t_k)) = \alpha_{p,k} < \alpha_{p,k-1}$
7:     Adjust the parameters so that $V(X_p(t_i)) = \alpha_{p,i}$ holds for all $p, i \leq k$
8:     **if** $V'(X) < 0$ for all X **then**
9:         Break For
10:     **end if**
11: **end for**

Using the algorithm, [12] gives analytic form of Lyapunov functions for given dynamics. An

example that [12] gives is finding the Lyapunov function of

$$\begin{cases} x' = -x + y, \\ y' = (x + y)\sin(x) - 3y. \end{cases} \tag{5}$$

At equilibrium $(0, 0)$. They choose a quadratic polynomial

$$V(x, y) = a_1 x^2 + a_2 xy + a_3 y^2,$$

as a candidate Lyapunov function, and train with algorithm 1 to find the appropriate coefficients a_1, a_2, and a_3. After 25 iterations they find the appropriate values to be $a_1 = 0.807, a_2 = -0.0581, a_3 = 1.0144$, and numerically show that the obtained Lyapunov function satisfies positive definiteness and the directional derivative satisfies negative definiteness.

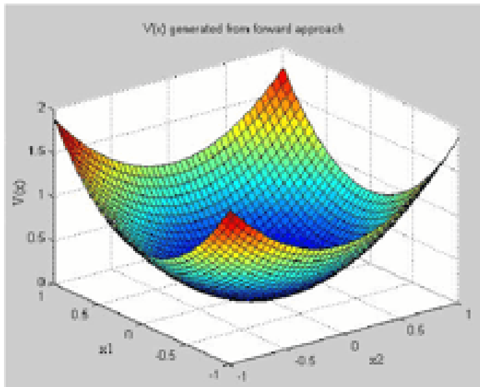
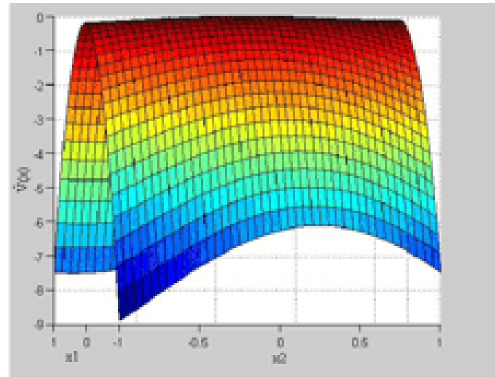


Fig. 2. The constructed Lyapunov function for the system (4-1)



ig. 3. The time derivative of the constructed Lyapunov functio

Figure 2: Learned Lyapunov function and its directional derivative of system (5) (Figure from [12])

Though the approaches in [11] or [12] gives analytic form expressions of the Lyapunov function, the method is not generalizable to complex dynamics, i.e. dynamics that does not have Lyapunov functions of polynomial expression, or of a single layered neural network. Also, these methods are not scalable, as the number of data needed grows exponentially with the dimension of states. Modern approaches mostly aim to address these issues, by using deep neural networks and inventing new techniques that make the training more stable and scalable.

2.1.4 Modern approaches

After the tremendous success of deep learning, using neural networks to approximate Lyapunov functions have recently gained much attention. The most prominent work is [6], which presents a modern baseline of approximating Lyapunov functions with neural networks and applying them in the field of control. In [6], the authors define a so-called 'Lyapunov-risk', which is exactly as the naive loss function that was introduced in Section 2.1.2. Then they jointly train a neural network $V(X; \theta)$ and a feedback controller $u(X)$. The parameters θ and u is updated so that the overall Lyapunov risk

$$L(\theta, u) = \frac{1}{n}\sum_{i=1}^{n}(V(0;\theta)^2 + \max\{0, -V(X_i;\theta)\} + \max\{0, \nabla_X V(X_i;\theta) \cdot F_u(X_i)\}),$$

for dynamics with control $X' = F_u(X)$ is minimized. Thus, when the learning is finished, $V(X;\theta)$ becomes the Lyapunov function of the dynamics $X' = F_u(X)$ for properly learned control function $u(X)$, which means that $u(X)$ is a control input that makes $X = 0$ asymptotically stable.

In addition to learning, the authors use a 'falsifier', which is a module that finds counterexamples on the state space that violates the Lyapunov conditions. The Lyapunov falsification constraint is given as a first-order boolean expression:

$$\Phi_\epsilon(X) = (\|x\| > \epsilon) \wedge (V(X) \leq 0 \vee DV(X) \geq 0), \tag{6}$$

Here, the condition $\|x\| > \epsilon$ exists only for numerical concerns, and the latter term of the boolean expression is merely a negation of the Lyapunov conditions. The solution to the boolean constraint can be found by SMT solvers such as dReal([6]). The overall algorithm can be found at algorithm 2.

Algorithm 2 Neural Lyapunov Control([6])

1: **function** LEARNING(X,f,q^{lqr}):
2:     Initialize the feedback controller u to $q^l qr$
3:     **for** $p = 0, 1, 2, ...$ **do**
4:         Compute Lyapunov risk $L(\theta, u)$
5:         Update the parameters θ, u with stochastic gradient descent
6:     **end for**
7:     **return** V_θ, u
8: **end function**
9: **function** FALSIFIER(f,u,V_θ,ϵ,δ):
10:     Encode conditions using (6)
11:     Using SMT solver with δ to verify the conditions
12:     **return** satisfiability
13: **end function**
14: **function** MAIN( ):
15:     **while** Satisfiable **do**
16:         Add counterexamples to X
17:         $V_\theta, u \leftarrow Learning(X, f, q^{lqr})$
18:         CE $\leftarrow$ Falsifier(f,u, V_θ, ϵ, δ)
19:     **end while**
20: **end function**

With the training process and falsifier module, [6] succeeds in finding stable control for problems such as inverted pendulum, wheeled vehicle path following, N-link planar robot balancing, with larger region of attraction compared to classical methods using SOS/SDP techniques.

[6] led to follow-up works that aims to improve the method by modifying the structure of learning or by improving the verification process([13],[14],[15]). In [13], the authors employ some minor changes to the method in [6] to improve performance. First, the authors use leaky-Relu functions instead of Relu($\max\{x, 0\}$) functions in their loss. To enforce the conditions $V(X) \geq 0$ and $DV(X) < 0$, they use the leaky-relu function $LR(p, a)$ defined as

$$LR(p,a) = \begin{cases} p & \text{if } p \geq 0, \\ ap & \text{otherwise,} \end{cases}$$

and define the Lyapunov risk as

$$L(\theta, X) = \frac{1}{n}\sum_{i=1}^{n} LR(DV(X_i) + \epsilon, a) + \frac{1}{n}\sum_{i=1}^{n} LR(-V(X_i) + \epsilon, a).$$

Using the leaky-relu function is beneficial, as the function induces training also when the constraints are satisfied, leading to an improvement in learning and a more numerically robust candidate Lyapunov function([15]). They also divide the boolean condition in (6) to two clauses, and erase the $\|x\| > \epsilon$ term to enforce full asymptotic stability near the origin. At last, when the counterexamples are generated, points in the neighborhood of each counterexample is also added to the overall dataset, which leads to a faster communication between the learner and falsifier. By using the techniques above, the authors are able to learn Lyapunov functions for systems that do not admit a global polynomial Lyapunov function faster than the existing approaches. They are also able to learn Lyapunov functions that cope with a larger domain than the previous approaches.

Some approaches aim to find the counterexamples by transforming the problem into equivalent optimization problems([14], [15]). In [14], the authors concentrate on piecewise linear dynamics, and aim to find a piecewise linear Lyapunov function for the dynamics. Similar to the approach in [6], they use the concept of Lyapunov risk and falsifiers. The Lyapunov risk defined in [14] is

$$L(\theta, X_1, X_2) = w_1 \sum_{i=1}^{n} \max\{0, \epsilon_1 |X_{1,i}| - V(X_{1,i}; \theta)\} + w_2 \sum_{i=1}^{n} \max\{0, DV_\theta(X) + \epsilon_2 V(X_{2,i}; \theta)\},$$

which is used to enforce $\epsilon_1 |X| \leq V(X;\theta)$ and $DV(X;\theta) \leq -\epsilon_2 V(X;\theta)$. The condition $V(0) = 0$ is automatically satisfied, as they use Relu activation without bias. Here, ϵ_1 and ϵ_2 are hyperparameters that are set at the beginning of learning. Different from [14], they use Mixed Integer Linear Programming to find the counterexamples instead of SMT solvers. For an appropriate domain $\mathcal{S}$ that includes the equilibrium $X = 0$, they define two different optimization problems:

$$\max_{X \in \mathcal{S}} \epsilon_1 \|X\| - V(X;\theta) \quad \text{and} \quad \max_{X \in \mathcal{S}} DV(X;\theta) + \epsilon_2 V(X;\theta).$$

We can change the problem into MILP form as $V(X;\theta)$ is piecewise linear and $\mathcal{S}$ is a polytope. By solving the optimization problem, we can obtain the values of X that violates $\epsilon_1|X| \leq V(X;\theta)$ or $DV(X;\theta) \leq -\epsilon_2 V(X;\theta)$. By using MILP to find the counterexamples, the authors are able to find appropriate piecewise-linear Lyapunov functions faster than the previous approaches. In [15], the authors formulate the counterexample finding problem into an equivalent MILP problem as in [14], but they use a feedback controller parametrized by a neural network.

In addition to works aiming to improve neural Lyapunov control, there has been works to fundamentally improve the learning process by cleverly designing the network architecture, and provide guarantees for the approximation power of the proposed architecture using tools from function approximation theory([16],[17]). [16] suggests a new architecture that the positive definiteness of $V(X;\theta)$

is guaranteed. Specifically, they suggest

$$V(X;\theta) = |\Phi_\theta(X) - \Phi_\theta(0)| + \bar{\alpha}\,\|x\|\,, \tag{7}$$

as a new parametrization for neural Lyapunov functions. Using the parameterization, two conditions $V(0;\theta) = 0$ and $V(X;\theta) \geq 0$ are automatically satisfied, and the Lyapunov risk is simplified to

$$L(\theta, X) = \frac{1}{n}\sum_{i=1}^{n} \max\{DV(X_i) + \gamma\,\|X_i\|\,, 0\}^2, \tag{8}$$

to enforce only the negative orbital derivative condition. With simplified risk, the learning procedure becomes comparably faster than other existing deep Lyapunov methods(Figure 3). Moreover, by using the algorithm in [16], the authors are able to train Lyapunov functions of systems with dimension up to 30.

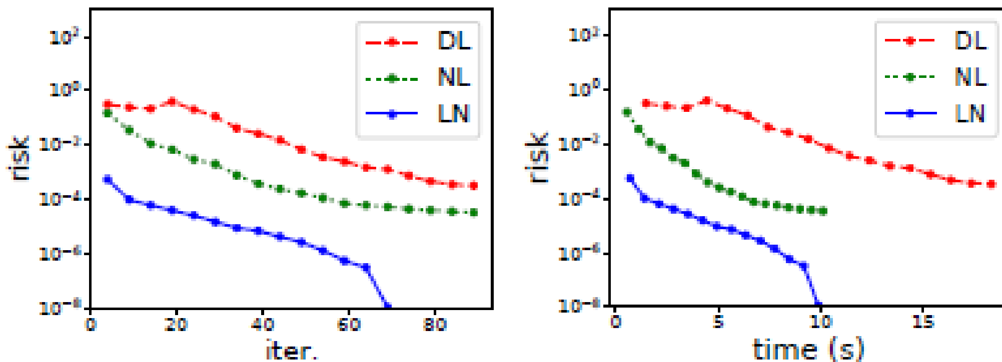


Figure 3: Learning procedure of different algorithms(Figure from [16]). The Blue line shows how risk decreases when using the algorithm proposed in [16]. The Red line is when using method from [17], and the green line is when using method from [6].

In addition to the empirical improvements by changing how $V(X;\theta)$ is parametrized, the authors also provide a proof for the guarantee to find a Lyapunov function when the size of the data points are sufficiently large. Specifically, the authors prove that for properly chosen γ and number of data proportional to $O(\epsilon^{-d})$, the minimizer of the proposed Lyapunov risk (8) can be approximated by the parametrization of (7).

Another modern approach exploits 'small gain theory' to design a novel architecture of neural networks to approximate Lyapunov functions([17]). The author aims to resolve curse of dimensionality, i.e. the phenomenon where the number of parameters needed to approximate a function increasing exponentially with the increase in dimension, by decomposing the Lyapunov function to s sub-Lyapunov functions.

$$V(X;\theta) = \sum_{i=1}^{s} \hat{V}_i(X;\theta_i).$$

For dynamics where small gain theory is applicable, it is realistic to assume such decomposable V([17]).

Then, the author uses s sub-networks to approximate each $\hat{V}_i(X;\theta_i)$. The curse of dimensionality is mitigated, as the needed number of parameters follow $O(\epsilon^{-d_{max}})$ instead of $O(\epsilon^{-d})$(when the dimension of ith subsystem is d_i and d_{max} denotes the maximum of $d_1, d_2, ..., d_m$). With the proposed method, [17] succeeds in finding the Lyapunov function for the 10-dimension dynamics given by the vector field

$$\hat{f}(X) = \begin{bmatrix} -x_1 + 0.5x_2 - 0.1x_9^2 \\ -0.5x_1 - x_2 \\ -x_3 + 0.5x_4 - 0.1x_1^2 \\ -0.5x_3 - x_4 \\ -x_5 + 0.5x_6 + 0.1x_7^2 \\ -0.5x_5 - x_6 \\ -x_7 + 0.5x_8 \\ -x_9 + 0.5x_{10} \\ -0.5x_9 - x_{10} + 0.1x_2^2 \end{bmatrix}.$$

It is clear that the dynamics is consisted of five independent dynamics that are asymptotically stable, along with four nonlinearities with small gains. By small gain theory, we can assume that the Lyapunov function will be decomposable with 5 sub-Lyapunov functions, and by a neural network architecture that computes each sub-Lyapunov function and summing them, we can approximate an appropriate Lyapunov function with much less parameters than $O(\epsilon^{-d})$.

2.2 Symbolic regression

Symbolic regression is an approach to machine learning in which both the parameters and the structure of an analytical model are optimized([18]). To explain further, it is a machine learning algorithm that learns the analytic expression of a given function $f : \mathbb{R}^n \to \mathbb{R}$, when data $\{(x_i, f(x_i)) \| \ i = 1, 2, ..., N\}$ is given. In this chapter, I will introduce different methods for symbolic regression, and present previous works that uses symbolic regression to generate Lyapunov functions with analytic form. In Section 2.2.1, I will introduce different methods to accomplish symbolic regression. In Section 2.2.2, I aim to go through works which use genetic algorithms to automatically find the analytic form of Lyapunov function.

2.2.1 Different methods of symbolic regression

The classical methods to perform symbolic regression are based on genetic programming. The first paper to apply genetic programming to symbolic regression dates back to 1994([19]), which is referred to as 'Koza-style symbolic regression'. The method encodes each symbolic expression into a tree structure(see Figure 4), where the leaf nodes denote the symbols and constant expressions, and the inner nodes denote the operation such as $+, -, *, /$. With the encoding, a standard genetic programming algorithm is run to find the best fitting expression for a given input-output pair of data.

The procedure of genetic programming is as following: First, we generate an initial population of

random expressions(thus, trees) composed of the primitive functions(such as $+, -, *, /, \exp$) and terminals(they may be inputs/constants) of the problem. This becomes generation 0 of the expressions. Then, we calculate the fitness of each expression in generation 0. The fitness function can be defined according to the objective of learning, and the paper uses MSE error to fit a given data. After that, the expressions at generation 0 are used to create new expressions for generation 1, by going through 'reproduction' or 'crossover' with the probability proportional to the fitness they have. Reproduction is passing the exact same expression through the next generation, and crossover is creating offsprings that inherits partial traits from both parent expressions by exchange subtrees of parent expressions. An example of crossover can be found in Figure 4. Here, the two expressions $0.234Z + X - 0.789$ and $ZY(Y + 0.314Z)$ are parent expressions, and offspring expressions $Y + 0.314Z + X - 0.789$ and $0.234Z^2Y$. are generated by exchanging the subtrees $0.234Z$ and $Y + 0.314Z$.

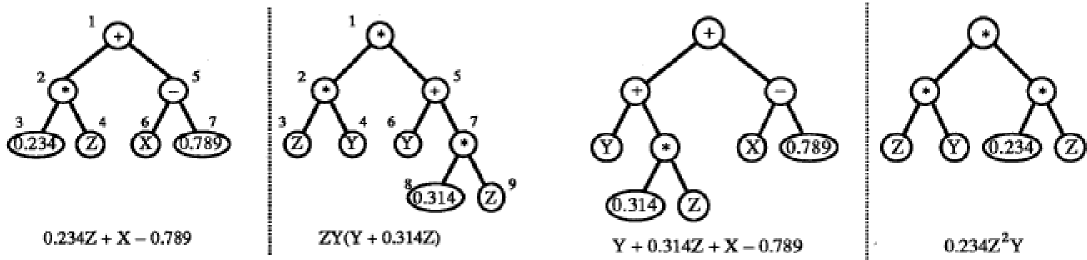


Figure 4: Example of crossover (Figure from [19])

The procedure is repeated until the algorithm arrives maximum generation, or the fitness has reached maximum value. As reproducing and crossover happens proportional to the fitness, the averaged fitness increases as generations pass. Also, the probability to find the best candidate for a given function increases as generations pass. [19] gives successful results on finding a closed-form expression of a 11-multiplexer and also presents an application in economy.

Previous works on symbolic regression aimed to improve the genetic programming based approach ([20],[21],[22]). [20] uses the concept of 'age' to advance genetic programming. Each expression has its own age, which is the number of generations it has passed through. The expressions are divided into several groups regarding their age, and crossover only happens for expressions within the same group or the group right before. Also, when an expression is moved from the youngest group, new expressions randomly replace the old ones. By restricting the competition within the expressions with same age group and regularly introducing new expressions, [20] is able to solve the problem of premature convergence. [21] goes a step further: instead of regularly advancing the expressions by age, the expressions move on a 2-dimensional plane, where the x-axis and the y-axis denotes the fitness and age of the expression, respectively. Thus, in this scheme, even though two expressions have different age they may compete according to their fitness. At last, [22] points out that using the MSE loss as a sole metric to calculate the fitness of expressions may penalize certain expressions that are particularly beneficial for some inputs, but behave poorly on average. The authors explicitly exploit those expressions by selecting the best candidates for each possible input and using them as parent expressions to create offspring for the latter generation.

Though genetic programming can be used as a general tool to find symbolic expressions from

a given dataset, it is not perfect. One serious drawback is that genetic programming tends to give expressions that are overly complicated, thus making the expression less interpretable. Some more recent works aim to find expressions that are less complex, by using ideas from Bayesian statistics or expected property of the expressions ([23],[24],[25]). [23] suggest a new method to solve symbolic regression problems with a Bayesian framework. They encorporate prior knowledge to improve effectively fitting symbolic regression, along with assuming the resulting formula as a fixed sum of sub-expressions. Using their framework results in comparably more simple expressions than the ones generated by genetic programming, and the method also needs much less memory. An example of generated expressions using both genetic programming and bayesian symbolic regression is as Figure 5. It is clear that bayesian methods are superior to genetic programming regarding the simplicity of generated expressions.

Table 3: Typical Expressions

Task		Expressions
f_1	Truth	$f_1 = 2.5x_0^4 - 1.3x_0^3 + 0.5x_1^2 - 1.7x_1$
	GP	$y = ((exp((\frac{-x_0}{0.80} + 0.81)) - (((\sin((0.80x_0)^2)$ $- \cos(x_1)^6) + \sin((0.80x_0)^2)) + \cos(x_1)))$ $-(((\frac{x_0}{-0.80}) + ((((\frac{x_0}{-0.80}) + \cos(x_1)) + ((\sin((0.71x_0)^2)$ $-((\sin(((0.71x_0))^2) - 0.77))^2) + 1.0)) + x_1))$ $+(0.76 + x_1))) + (((\frac{x_0^2}{0.78}))^2 + \frac{x_0^2}{0.80})$
	BSR	$y = (-0.02) + (-1.30)[x_0^3 + 1.30x_1 + 0.09]$ $+(0.49)[5.05x_0^4 + x_1^2 + 0.31]$
f_2	Truth	$f_2 = 8x_0^2 + 8x_1^3 - 15$
	GP	$y = (exp(1.82)x_1^3) + 5.26(x_0^2 - (\cos((0.90x_0))$ $*(exp(0.187) + \cos((x_0^2 \cos(0.75)))))$ $+(x_1 - 0.77)^3 + exp(x_1 - 0.38)(x_1 - 0.38)$
	BSR	$y = (-0.02) + (-1.38)[-7.56x_0^2 + 2.85]$ $+(8.00)[-0.30x_0^2 + x_1^3 - 1.38]$

Figure 5: Expressions created by genetic programming and bayesian symbolic regression (Figure from [23])

[24] and [25] presents 'AI-Feynman' and 'AI-Feynman 2', a novel symbolic regression algorithm inspired from physics. The algorithm is motivated by the fact that many expressions that show the laws of nature have properties that help them easier to be discovered. The properties they suggest are: (1) Dataset often have physical units that can be used to help symbolic regression. (2) The expressions are often composed of lower-degree polynomials. (3) The expressions are often composition of simpler modules. (4) The expressions are analytic. (5) The expressions are symmetric. and (6) The expressions are often separable with different variables. The algorithm exploits these facts by first testing if the expression can be written in a simple form, by dimension analysis, testing polynomial regression, and using brute force to find simple expression of the given function. If the methods fail, the algorithm approximates the mapping with a neural network, and check if the mapping has properties such as compositionality, symmetricity, or separability. If such properties hold and the mapping that we want to express can be decomposed into smaller modules, the algorithm recursively works on each module to find an expression for individual modules, and then merge them into one.

To check the modular properties of a given mapping, [25] uses gradients of the neural network. Here, I will give an example of how the model notices submodules of the mapping. Suppose $f : \mathbb{R}^d \to \mathbb{R}$ is composed of two mappings $g : \mathbb{R} \to \mathbb{R}$ and $h : \mathbb{R}^d \to \mathbb{R}$, i.e. $f(x) = g(h(x))$ holds. Then, when we check the gradient, we can easily see that $\nabla f(x) = g'(h(x))\nabla h(x)$ holds. Thus, the two gradients $\nabla f(x)$ and $\nabla h(x)$ are parellel, and finding a function $h(x)$ that has gradient parellel to the mapping $f(x)$ leads to the compositionability of f. As we have no direct access to f or h, we approximate them with neural networks and analyze gradients of the neural networks instead.

With the new algorithm, AI-Feynman is able to find symbolic expression to all 100 equations from the *Feynman Lectures on Physics*, and successfully expresses 90% of the bonus problems, where the previous commonly used algorithms such as Eureqa only succeeds in 15% of the problems. The succeeded expressions include very complicated expressions such as

$$v = c\frac{(v_1 + v_2 + v_3 + v_4)/c + (v_2v_3v_4 + v_1v_2v_3 + v_1v_2v_4 + v_1v_3v_4)/c^3}{1 + (v_1v_2 + v_1v_3 + v_1v_4 + v_2v_3 + v_2v_4 + v_3v_4)/c^2 + v_1v_2v_3v_4/c^4},$$

which is the sum of velocity in the sense of relativity. This amazing success shows that the proposed algorithm is effectively finding expressions that are both simple and fit the data well even for challenging problems.

2.2.2 Finding symbolic expression of Lyapunov Functions

Using genetic programming to find Lyapunov functions is intuitive. Instead of using the MSE loss as the fitness function, using a fitness function to encourage the expressions to satisfy the Lyapunov conditions is possible. [26] uses two different fitness function to enforce the Lyapunov conditions. The first function is given as

$$f_i = -r_0 2^{-r_0(D_+ + V_-)}, \tag{9}$$

for expression i. Here, r_0 denotes the radius of the ball where the conditions of Lyapunov functions will be checked. For grid points inside the ball, D_+ denotes the number of points that violate the negative semidefiniteness of directional derivative, and V_- denotes the number of points that violate the positive semidefiniteness of the expression. The genetic algorithm aims to decrease the fitness function in order to satisfy the Lyapunov constraints. Next, the second function

$$\hat{f}_i = \frac{1}{1 + 1/r_i}, \tag{10}$$

is used to assess the learned expression. Here, r_i is the minimum distance between the origin and the maximal sublevel set $L_v(c)$ where Lyapunov conditions are satisfied. r_i is used as an index to obtain information of the quality of the learned expression's region of attraction. Thus, with fitness functions 9 and 10, the genetic algorithm generates Lyapunov functions with large region of attraction.

However, the learned Lyapunov functions are very complex even for simple dynamics. For example, the learned Lyapunov function of

$$\begin{cases} x' = -y, \\ y' = x - y + x^2 y. \end{cases} \tag{11}$$

is given as the expression

$$V(x,y) = 1.5124x^2 + 1.2(0.6246x + y - 1.5159\sin(1.0521x))^2,$$

which is almost impossible to be used to prove a stability of $(0,0)$.

To ameliorate such drawback of genetic programming, the authors in [27] uses grammatical evolution to find symbolic expression of Lyapunov functions. Grammatical evolution is a method that uses 'encodings' rather than 'expressions' as individuals in a population. When we decode the encoding sequence with given grammar rules, we can obtain the expression corresponding to the encoding. As grammatical evolution is known to produce less complicated answers than genetic programming, the authors use it to find an analytic form of Lyapunov functions. They are able to find a simple form of Lyapunov function for dynamics such as

$$\begin{cases} x' = y, \\ y' = -x - x^2 y. \end{cases} \tag{12}$$

where the learned Lyapunov function is

$$V(x,y) = x^2 + y^2.$$

With the learned Lyapunov function, it is possible to prove that $(0,0)$ is asymptotically stable by using LaSalle's invariance principle. However, there is no analysis on higher dimension dynamics in [27].

3 Experiments

In this Section, I will first describe the novel algorithm that is used to find an analytic form of the Lyapunov function for a given equilibrium of a dynamic system. Then I will briefly provide the details of the experimental settings, and provide various examples where the proposed method succeeds in finding the Lyapunov function for the equilibrium.

3.1 Algorithm

The algorithm is divided into two steps: Learning the Lyapunov "function", and learning the analytic form of that function. For the first step, I used the approach in [16] due to its simplicity in the loss function and its ability to approximate Lyapunov functions of very high dimensional dynamics. However, I used a slightly different structure for parametrization of Lyapunov functions,

$$V(X;\theta) = (\Phi_\theta(X) - \Phi_\theta(0))^2 + \bar{\alpha}\left\|x\right\|^2,$$

for a neural network $\Phi_\theta(X)$. The modification was made to exploit the gradient method in pytorch. For the training the empirical loss

$$L(\theta, X) = \frac{1}{n}\sum_{i=1}^{n} \max\{DV(X_i) + \gamma\left\|X_i\right\|, 0\}^2,$$

was used, identical to [16]. Batchwise gradient descent was applied during learning, and Adam optimizer was used to train the neural network.

For the second step, I sampled the points $(X_i, V(X_i, \theta))$ where X_i are points near the equilibrium to train the symbolic regression algorithms. I used two different symbolic regressors: Pysr([28]), a genetic algorithm based python symbolic regression package, and AIFeynman([24]), a brute-force based symbolic regression algorithm. Pysr was used at first to find simple relationship between X_i and $V(X_i, \theta)$, and AIFeynman was used to find further more complicated relation that Pysr fails to discover.

The summary of the proposed method is as Algorithm 3.

Algorithm 3 Proposed Algorithm

Require: N, M, lr, $\bar{\alpha}$, γ, B
1: **for** $p = 0, 1, 2, ..., N-1$ **do**
2: Sample data $(X_p, F(X_p))$ for a given dynamics near equilibrium
3: **end for**
4: **while** $L(\theta, X) > 1e-4$ **do**
5: Construct a batch X_B of size B
6: $\theta \leftarrow AdamOptimizer(\theta, lr, \nabla_\theta L(\theta, X_B))$
7: **end while**
8: **for** $p = 0, 1, 2, ..., M-1$ **do**
9: Sample data $(X_p, V(X_p; \theta))$ for training of symbolic regressors
10: **end for**
11: Use Pysr to find an analytic form of $V(X_p; \theta)$
12: **if** Pysr fails **then**
13: Use AIFeynman to find an analytic form of $V(X_p; \theta)$
14: **end if**

3.2 Experimental settings

All experiments were done using Pytorch in Python 3.9 with processor AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz, and a neural network of depth 5 and width 10 was used for all experiments. For Adam optimizer I used a learning rate of $1e-3$ and $\beta_1 = 0.9$, $\beta_2 = 0.999$, and used a batch size $B = 32$, number of data for training symbolic regression as $M = 1000$. The parameters $N, \bar{\alpha}, \gamma$ were crucial to training, and I had to use different values for different problems. The values of the hyperparameters for each experiment can be found in table 1.

Dynamics	N	$\bar{\alpha}$	γ
2D Harmonic Oscillator	12800	0.1	0.01
Pendulum	128000	0.1	0.1
Damped Pendulum	12800	0.01	0.1
Example 2 from [27]	12800	0.01	0.01
2D Curve track from [16]	12800	0.1	0.1
$\sin(x)\sin(y)$ example	12800	0.1	0.1
Lorenz	25600	0.01	0.1
High order harmonic oscillator(4D)	128000	0.1	0.1

Table 1: Hyperparameters for each experiments

A thorough explaination on each dynamics will be given in the next Section.

3.3 Examples

In this Section I will show various applications of the proposed algortihm, and how it can be used to prove the asymptotic stability of a given equilibrium

3.3.1 Classic 2D examples

Let's start with the harmonic oscillator, perhaps one of the most simple 2 dimensional dynamics. 2D harmonic oscillator is given as

$$\begin{cases} x' = y, \\ y' = -x. \end{cases}$$

By the training process, we can obtain the Lyapunov function (but not knowing the analytic form) as below. We can either use Pysr or AIFeynman to find the symbolic representation of the Lyapunov function. By using AIFeynman, we can obtain that the learned function $V(x, y) = x^2 + y^2$.

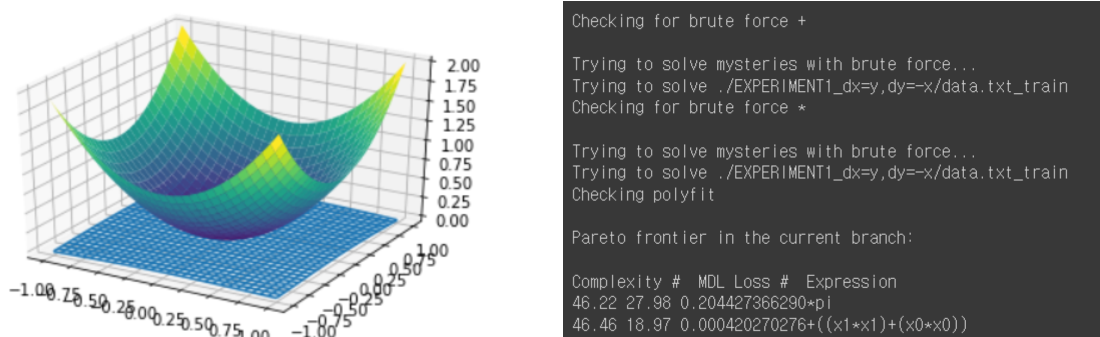
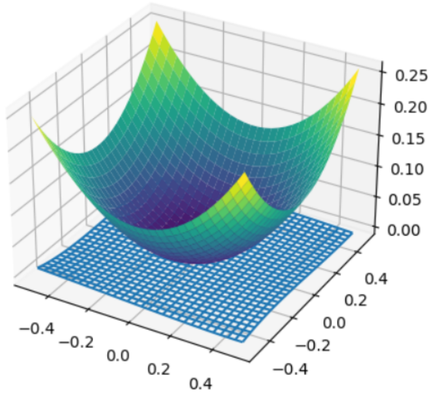


Figure 6: Learned Lyapunov function and its symbolic regression for 2D Harmonic Oscillator. The left Figure shows the learned Lyapunov function, and the right Figure shows the AIFeynman window

Another classical example from physics is the pendulum. The dynamics of the pendulum is

$$\begin{cases} x' = y, \\ y' = -\sin(x). \end{cases}$$

which is almost identical to the harmonic oscillator. However, the Lyapunov function(or the conserved energy) of the dynamics is $V(x, y) = \frac{1}{2}y^2 + (1 - \cos(x))$, which is not a polynomial thus cannot be solved by classical methods([11],[12]) that assumes very specific simple form of Lyapunov functions. However, using the proposed method we can both find a Lyapunov function of the given dynamics and its closed form identical to the one above. This implies that our method is more versatile, and can be more widely used to a variety of complex dynamics.



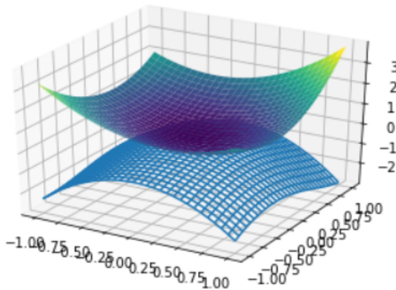
Complexity	Loss	Equation
1	24.99426	8.192031
3	24.86929	(x1 + 8.1691065)
4	24.626	(exp(x1) + 7.1293173)
5	21.29388	(x0 * (x0 * 76.252914))
6	13.23794	((cos(x0) * -95.75766) + 100.031685)
8	7.770871	((cos(x1 * x0) * -923.20667) + 928.55505)
9	0.033197	((cos(x1) * (cos(x0) * -107.51576)) + 107.29689)
11	0.015584	((cos(x1) * -104.25412) + ((cos(x0) * -100.66018) + 204.8908))
12	0.008596	((cos(x0) * -102.442696) + (((x1 * x1) * 51.490158) + 102.39776))
14	0.002883	((cos(x0) * -102.43041) + ((((x1 + 0.005275002) * x1) * 51.41395) + 102.385956))
15	0.002883	((cos(x0) * -102.43041) + ((((x1 + sin(0.005275002)) * x1) * 51.41395) + 102.385956))
16	0.002189	((cos(x0) * -102.43327) + (((x1 * x1) * (51.405354 + (x1 * 1.8179433))) + 102.38903))
17	0.002182	((cos(x0) * -102.43327) + (((x1 * x1) * (51.405354 + (sin(x1) * 1.9057959))) + 102.38903))
18	0.001994	((cos(x0) * -102.43327) + (((x1 * x1) * ((51.405354 + x0) + (x1 * 1.9514858))) + 102.38903))
19	0.001976	((cos(x0) * -102.43327) + (((x1 * x1) * ((51.405354 + sin(x0)) + (x1 * 1.9514858))) + 102.38903))
20	0.001888	((cos(x0) * -102.43327) + (((x1 * x1) * ((51.405354 + (x1 + x1)) + (x0 * 0.6574972))) + 102.38903))

Figure 7: Learned Lyapunov function and its symbolic regression for Pendulum. The left Figure shows the learned Lyapunov function, and the right Figure shows list of possible functions suggested by Pysr

And last, let's think about the damped pendulum. The dynamics of the damped pendulum is

$$\begin{cases} x' = y, \\ y' = -y - \sin(x). \end{cases}$$

and is known to have an asymptotically stable equilibrium $(0, 0)$. In this case, $x^2 + y^2$ fails to work as a strict Lyapunov function, and it becomes a tedious job to find a strict Lyapunov function of the dynamics. However, by using the proposed algorithm, we can simply check that the asymptotic Lyapunov function can be approximated as $V(x, y) = 2x^2 + xy + 2y^2$.




```
Checking for brute force +

Trying to solve mysteries with brute force...
Trying to solve ./EXPERIMENT3_dx=y,dy=-y-sinx/data.txt_train
Checking for brute force *

Trying to solve mysteries with brute force...
Trying to solve ./EXPERIMENT3_dx=y,dy=-y-sinx/data.txt_train
Checking polyfit

Pareto frontier in the current branch:

Complexity #  MDL Loss #  Expression
46.64 28.45 0.273342591097*pi
51.71 27.96 0.454513496847+(x0*x0)
55.25 27.72 1.674755268141*(x0*(x1+x0))
58.93 27.7 1.754527395904*(x0*(x1+sin(x0)))
113.5 27.67 -3.970514737545*(cos((x0/cos(x1)))-1)
444.99 22.18 0.0519550189690838*x0**3 + 0.105223518477078*x0**2*x1 + 1.49402117755773*x0**2 + 0.105854987372479*x0*x1**2 + 0.834569686346491*x0*x1 + 0.00289660376659395*x0 +
0.0528953445310995*x1**3 + 1.36153184866285*x1**2 + 0.00187512681396736*x1 - 5.27817969703668e-5
```

Figure 8: Learned Lyapunov function and its symbolic regression for Damped Pendulum. The left Figure shows the learned Lyapunov function, and the right Figure shows the output of AIFeynman

Moreover, we can use the obtained Lyapunov function to prove asymptotic stability of $(0, 0)$. Using the function, the directional derivative $DV(x, y)$ becomes

$$DV(x, y) = y(4x + y) - (y + \sin(x))(4y + x) = 3xy - 3y^2 - 4y\sin(x) - x\sin(x).$$

Showing that $(0,0)$ is a local maximum of $DV(x,y)$ suffices to show that $(0,0)$ is asymptotically stable(and $V(x,y)$ is the strict Lyapunov function). Let's use the second derivative test. We know that

$$\nabla DV(x,y) = \begin{bmatrix} -(x+4y)\cos(x) - \sin(x) + 3y \\ 3x - 4\sin(x) - 6y \end{bmatrix},$$

and

$$\nabla^2 DV(x,y) = \begin{bmatrix} 4y\sin(x) + x\sin(x) - 2\cos(x) & 3 - 4\cos(x) \\ 3 - 4\cos(x) & -6 \end{bmatrix}.$$

Thus,

$$\nabla DV(0,0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

and

$$\nabla^2 DV(0,0) = \begin{bmatrix} -2 & -1 \\ -1 & -6 \end{bmatrix}.$$

$\nabla^2 DV(0,0)$ has eigenvalues $-4+\sqrt{5}$, $-4-\sqrt{5}$, which makes it negative definite. This means $DV(x,y)$ becomes negative definite, $V(x,y)$ a strict Lyapunov function, and $(0,0)$ an asymptotic stable equilibrium. This example shows that our proposed method can help mathematicians obtain a strict Lyapunov function in an analytic form and prove the asymptotic stability of an equilibrium.

3.3.2 Some harder examples

We provide further examples to illustrate two points made in the previous Section: The proposed method is versatile and can find Lyapunov functions not restricted to a polynomial, and can help find strict Lyapunov functions for systems that do not have trivial Lyapunov functions.

The first example is

$$\begin{cases} x' = -\sin(x)\cos(y) + \cos(x)\sin(y), \\ y' = -\sin(x)\cos(y) + \cos(x)\sin(y). \end{cases}$$

For equilibrium $(\frac{\pi}{2}, \frac{\pi}{2})$, we can obtain the Lyapunov function $V(x,y) = 2 - \sin(x) - \sin(y)$ using the proposed algorithm. By using a similar method done for the damped pendulum case, we can easily see that $V(x,y)$ is indeed a strict Lyapunov function. We can also intuitively check that near $(\frac{\pi}{2}, \frac{\pi}{2})$, $DV(x,y) < 0$ by using GeoGebra([29]). From Figure 9, we can see that near $(\frac{\pi}{2}, \frac{\pi}{2})$, the color of the Figure is gray, meaning that $DV(x,y) < 0$ in the area. We can analytically show that $(\frac{\pi}{2}, \frac{\pi}{2})$ is indeed an asymptotic stable point using the same argument as in the damped pendulum. I omit the proof here.

The second example is the example used in [27]. In [27] the authors present a synthetic dynamics

$$\begin{cases} x' = y^2 - \tan(x), \\ y' = -y + x. \end{cases}$$

Complexity	Loss	Equation
1	1.117557	1.721906
4	1.045081	exp(cos(sin(x1)))
5	0.680664	(sin(exp(x1)) + 1.987877)
6	0.553501	((sin(x1) * -20.979492) + 21.8291)
7	0.552299	((inv(sin(x1)) * 18.795975) + -17.917543)
8	0.552121	((sin(x1 + 0.006251398) * -20.987709) + 21.836548)
9	0.001732	(((sin(x0) + sin(x1)) * -20.98159) + 41.954273)
10	0.001617	(((sin(x1) + cos(cos(x0))) * -21.703499) + 43.380814)
12	0.001106	((((sin(x0) + sin(x1)) * -21.4886) + 41.967384) + sin(x0))
13	0.001104	((((sin(x0) + sin(x1)) * -21.454641) + 41.90032) + cos(cos(x0)))
14	0.000597	(((sin(x0 + (cos(x1) * -0.020489138)) + sin(x1)) * -20.968987) + 41.928875)
16	0.000597	((((sin(x0 + (cos(x1) * -0.020497357)) + sin(x1)) * -20.98153) + 41.95431) * 0.99917954)
17	5.14E-05	((((sin(x0 + (cos(x1) * -0.019182177)) + sin(x1)) * -21.454618) + 41.90028) + sin(x0))
18	4.17E-05	((((sin(x0 + (cos(x1) * -0.019182177)) + sin(x1)) * -21.454641) + 41.90028) + cos(cos(x0)))
19	4.17E-05	((((sin(x0 + sin(cos(x1) * -0.019182177)) + sin(x1)) * -21.454641) + 41.90028) + cos(cos(x0)))
20	3.64E-05	((((sin(x0 + (cos(x1) * -0.019817347)) + sin(x1 + 0.0006350411)) * -21.454618) + 41.90028) + cos(cos(x0)))

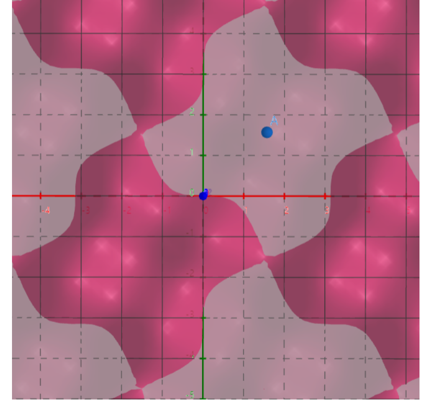


Figure 9: Learned Lyapunov function's symbolic regression and the plot of $DV(x,y)$ on GeoGebra

to assess their proposed algorithm, and find $V(x,y) = x^2 + y^2|y|$ as a candidate Lyapunov function. By using my algorithm, we can obtain $V(x,y) = x^2 + y^2$ as a strict Lyapunov function for the given dynamics. For details, see Figure 10

```
Checking for brute force +

Trying to solve mysteries with brute force...
Trying to solve ./EXPERIMENT2_dx=-tanx+y^2,dy=-y+x/data.txt_train
Checking for brute force *

Trying to solve mysteries with brute force...
Trying to solve ./EXPERIMENT2_dx=-tanx+y^2,dy=-y+x/data.txt_train
Checking polyfit

Pareto frontier in the current branch:

Complexity #  MDL Loss #  Expression
46.21 27.95 0.203161883992*pi
46.21 27.95 -0.203220707657*(-pi)
50.85 27.47 0.251102458905+(x1*x1)
54.48 27.36 0.982572256662*(x1*(x1+x0))
58.14 27.34 1.017694835331*(x1*(x0+sin(x1)))
59.44 27.26 1.253397464078+(-cos((x1+x0)))
59.76 26.92 -0.782744675939+(cos((x1/cos(x0))))**(-1)
59.76 26.92 -0.783155933145+(cos((x0/cos(x1))))**(-1)
63.72 25.27 -0.814359595311+((cos(x1)*cos(x0)))**(-1)
111.4 25.07 -1.914263400567*(cos(x1)-(cos(x0))**(-1))
262.22 22.16 0.987710015983576*x0**2 - 0.0146886092287513*x0*x1 - 0.00624802906800204*x0 + 0.988436876253326*x1**2 - 0.00367876088128615*x1 - 6.2366578
```

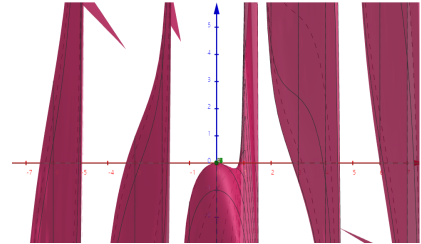


Figure 10: Learned Lyapunov function's symbolic regression and the plot of $DV(x,y)$ on GeoGebra, dynamics from [27]

And last, I applied the method to 2D curve example shown in [16]. The dynamics is given as

$$\begin{cases} x' = -\sin(y), \\ y' = x\cos(y) - \sin(y). \end{cases}$$

By using the algorithm, we can obtain $V(x,y) = 25x^2 - 17xy + 15y^2$ as a closed form of the strict Lyapunov function, and by GeoGebra we can check that it is indeed a strict Lyapunov function. For details, check Figure 12.

Complexity	Loss	Equation
1	4.719748	2.525856
3	4.631658	exp(exp(x0))
4	4.608452	(exp(x0) + 1.4876649)
5	2.845451	((x0 * x0) * 31.48914)
6	2.810892	(sin(x0) * (x0 * 32.449673))
7	2.195136	((x0 * (x0 * 23.48967)) + 1.0877349)
8	2.185769	((sin(x0) * (x0 * 24.30754)) + 1.0710491)
9	1.694786	(x0 * (((x1 * -0.5701548) + x0) * 32.12506))
10	1.663744	(x0 * (((x1 * -0.55257666) + sin(x0)) * 33.09545))
11	1.075059	((x0 * (((x1 * -0.7436798) + x0) * 24.304855)) + 1.0620092)
12	1.045789	((x0 * (((x1 * -0.74331665) + x0) * 24.622118)) + exp(x1))
13	0.844061	(((x1 * -0.50142145) + x0) * (((x1 * -0.5014124) + x0) * 27.579966))
14	0.821104	(((x1 * -0.5323286) + x0) * (sin((x1 * -0.5323286) + x0) * 27.579773))
15	0.010551	((x0 * (((x1 * -0.6912532) + x0) * 25.900492)) + (x1 * (x1 * 15.026508)))
16	0.010543	((x0 * ((sin(x1 * -0.6974558) + x0) * 25.9049)) + ((x1 * 15.026459) * x1))
17	0.006932	((x0 * (((x1 * -0.692947) + x0) * 25.91344)) + ((x1 * 15.034435) * (x1 + 0.015825817)))
19	0.004664	((x0 * (((x1 * -0.69125) + x0) * (x0 + 25.900469))) + (x1 * ((x1 * 15.026627) + 0.26892906)))

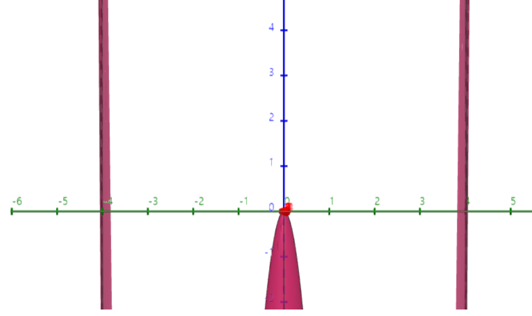


Figure 11: Learned Lyapunov function's symbolic regression and the plot of $DV(x, y)$ on GeoGebra, 2D curve example

3.3.3 Higher order examples

In this Section, I will present two higher-order dynamics where the algorithm successfully finds the Lyapunov function. The first example is rather simple, a 4D harmonic oscillator given as

$$\begin{cases} x' = y, \\ y' = -x, \\ z' = w, \\ w' = -z. \end{cases}$$

It is clear that $V(x, y, z, w) = x^2 + y^2 + z^2 + w^2$ is a Lyapunov function for the equilibrium $(0, 0, 0, 0)$. However, Pysr fails in finding the closed form, as genetic algorithms are prone to failure when there are too many variables. By using AIFeynman, we can successfully find V as above.

The second example is far more interesting. Here, we aim to analyze the stability of $(0, 0, 0)$ of a Lorenz system with small Rayleigh number. Specifically, we analyze the dynamics

$$\begin{cases} x' = 3(-x + y), \\ y' = \frac{1}{2}x - y - xz, \\ z' = -z + xy. \end{cases}$$

It is known that $(0, 0, 0)$ is an asymptotic stable point when the Rayleigh number is less than 1. To obtain the analytic form of Lyapunov function, I obtained two different data, $(x, z, V(x, 0, z))$, $(x, y, V(x, y, 0))$ and approximated each. By approximating, I could see that $V(x, 0, z) = 2.5(x^2 + xz + z^2)$ and $V(x, y, 0) = 2.3x^2 + 9xy + 20y^2$. With the result I got the candidate Lyapunov function

$$V(x, y, z) = x^2 + 8.5y^2 + z^2 + xz + 4xy.$$

It is clear that $V(x, y, z)$ is positive definite.

By using the candidate Lyapunov function, we can show that $(0, 0, 0)$ is indeed a strict Lyapunov function. By direct calculation one gets

Complexity	Loss	Equation
1	0.110161	0.375028
4	0.105911	sin(exp(cos(x1)))
5	0.084762	((x0 * x1) + 0.38085514)
6	0.078316	sin(exp(cos(x0 + x1)))
7	0.0595	(((x0 + x1) * x1) * 3.2984204)
8	0.03358	sin(exp(cos(x0 + x1) + 0.08901776))
9	0.026119	(((x0 + x1) * 1.9516882) * (x0 + x1))
10	0.018142	(exp((x0 + x1) * (x0 + x1)) + -0.81520027)
11	0.007404	(((x0 * (x0 + x1)) + (x1 * x1)) * 2.403987)
12	0.007222	(((sin(x0) * x0) + ((x0 + x1) * x1)) * 2.4350448)
13	0.004911	(((x0 * x0) + ((x1 + -0.05290573) * (x1 + x0))) * 2.4004748)
14	0.004754	(((x0 * sin(x0)) + ((x1 + x0) * (x1 + -0.051574793))) * 2.435277)
15	0.002775	(((x0 * x0) + ((x1 + x0) * x1)) * ((x0 * -0.8885982) + 2.402853))
16	0.002561	(((sin(x0) * x0) + ((x1 + x0) * x1)) * ((x0 * -0.9070986) + 2.4335086))
17	0.000898	((((x0 + x1) * ((x1 * -1.8881539) + 2.4180048)) * x0) + (x1 * (x1 * 2.3684354)))
18	0.000498	((((x0 + x1) * (exp(x1 * -1.7481914) + 1.2656101)) * x0) + (x1 * (x1 * 2.333022)))
19	0.000409	((((sin(x0) + x1) * (exp(x1 * -1.750961) + 1.3098282)) * x0) + (x1 * (x1 * 2.3467293)))
20	0.000353	((((x0 + x1) * (exp(x1 * -1.8096132) + 1.2429012)) * x0) + (x1 * ((x1 * 2.3506413) + 0.050792955)))

Complexity	Loss	Equation
1	2.194417	1.563457
4	2.090464	exp(x1 * -2.5482085)
5	0.631889	((x1 * x1) * 17.997408)
6	0.630051	((sin(x1) * x1) * 18.547277)
7	0.469318	((x1 * (x0 + x1)) * 14.650304)
8	0.440873	(((sin(x0) + x1) * x1) * 14.83036)
9	0.103057	((x1 * (x1 + (x0 * 0.48951036))) * 17.930195)
10	0.100934	((x1 * (sin(x1) + (x0 * 0.47512245))) * 18.47816)
11	0.057857	(((x1 + -0.036058832) * (x1 + (x0 * 0.48951036))) * 17.930195)
12	0.056037	((((x1 + x0) + x1) * (sin(x1) + -0.035919387)) * 9.216174)
13	0.04449	(((((x1 + x0) + x1) * (x1 + -0.03913385)) * 8.477429) + 0.16121653)
14	0.044124	(((((sin(x1) + x0) + x1) * (x1 + -0.038961083)) * 8.593449) + 0.15958247)
15	0.028259	((((x1 + (x1 + x0)) * (x1 + -0.03833892)) * 8.705275) + (x0 * x0))
16	0.027525	(((((x1 + x0) + x1) * (sin(x1) + -0.03682031)) * 8.998977) + (x0 * x0))
17	0.017606	((((x1 + (x1 + x0)) * (x1 + -0.038961083)) * 8.593449) + (x0 * (x0 + x0)))
18	0.016318	((((x1 + (sin(x1) + x0)) * (x1 + -0.038961083)) * 8.593449) + (x0 * (x0 + x0)))
20	0.016107	((((sin(x1) + (x1 + x0)) * (x1 + -0.038961083)) * 8.593449) + (x0 * ((-0.038961083 + x0) + x0)))

Figure 12: Learned Lyapunov function's symbolic regression, V(x,0,z) and V(x,y,0)

$$DV(x,y,z) = x^2y - 4x^2z - 4x^2 - 15xyz - 1.5xy - 4xz - 5y^2 + 3yz - 2z^2,$$

$$\nabla DV(x,y,z) = \begin{bmatrix} x(2y-8z-8) + y(-15z-1.5) - 4z \\ x^2 + x(-15z-1.5) - 10y + 3z \\ -4x^2 - x(15y+4) + 3y - 4z \end{bmatrix},$$

$$\nabla^2 DV(0,0,0) = \begin{bmatrix} -8 & -1.5 & -4 \\ -1.5 & -8 & 3 \\ -4 & 3 & -4 \end{bmatrix}.$$

The eigenvalues of $\nabla^2 DV(0,0,0)$ are approximately $-10.5417, -9.3466, -0.1116$ thus all smaller than 0, which makes $DV(x,y,z)$ negative definite, $V(x,y,z)$ a strict Lyapunov function, and $(0,0,0)$ an asymptotically stable equilibrium.

4 Conclusion

In this thesis I provided a survey on automatically finding Lyapunov functions of a given dynamic system, both with deep learning and symbolic regression. I also proposed a novel algorithm that can find the analytic form of the Lyapunov function without structural presumptions that works for both 2 and higher order dynamics.

Though the work has shown intriguing results on various dynamics, it is yet said to be perfect. For example, it fails when the Lyapunov function is too complicated: The Lotka-Volterra system given as the dynamics

$$\begin{cases} x' = x(1-y), \\ y' = y(-1+x). \end{cases}$$

has a constant of motion $V(x,y) = x + y - \ln(xy)$, but both Pysr and AIFeynman failed in finding the symbolic expression of the obtained Lyapunov function. Also, the deep learning process sometimes learned a rather complicated Lyapunov function. For instance, for the Lorenz example in Section 3.3.3, there actually is a simpler Lyapunov function $V(x,y,z) = x^2 + 3y^2 + 3z^2$. For further research,

improving the deep learning process to learn a more simple and less noisy function will be a promising way to elevate the performance of the given method.

Nevertheless, this work has shown possibility of implementing an algorithm that can help mathematicians prove the stability of a given equilibrium with direct Lyapunov method.

References

[1] Jiequn Han, Arnulf Jentzen, and Weinan E. "Solving high-dimensional partial differential equations using deep learning". In: *Proceedings of the National Academy of Sciences* 115.34 (2018), pp. 8505–8510.

[2] Jun-Ting Hsieh et al. "Learning neural PDE solvers with convergence guarantees". In: *arXiv preprint arXiv:1906.01200* (2019).

[3] Yuehaw Khoo, Jianfeng Lu, and Lexing Ying. "Solving parametric PDE problems with artificial neural networks". In: *European Journal of Applied Mathematics* 32.3 (2021), pp. 421–435.

[4] Patrick Kidger. "On neural differential equations". In: *arXiv preprint arXiv:2202.02435* (2022).

[5] Adam Zsolt Wagner. "Constructions in combinatorics via neural networks". In: *arXiv preprint arXiv:2104.14516* (2021).

[6] Ya-Chien Chang, Nima Roohi, and Sicun Gao. "Neural Lyapunov Control". In: *CoRR* abs/2005.00611 (2020). arXiv: `2005.00611`. URL: `https://arxiv.org/abs/2005.00611`.

[7] D.V. Prokhorov. "A Lyapunov machine for stability analysis of nonlinear systems". In: *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*. Vol. 2. 1994, 1028–1031 vol.2. DOI: `10.1109/ICNN.1994.374324`.

[8] P Werbos. "The brain as a neurocontroller: New hypotheses and new experimental possibilities". In: *Origins: Brain and Self-Organization (KH Pribram, ed.), Erlbaum* (1994).

[9] Gursel Serpen. "Empirical approximation for Lyapunov functions with artificial neural nets". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. IEEE. 2005, pp. 735–740.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[11] Vassilios Petridis and Stavros Petridis. "Construction of neural network based lyapunov functions". In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings.* IEEE. 2006, pp. 5059–5065.

[12] Navid Noroozi et al. "Generation of lyapunov functions by neural networks". In: *Proceedings of the World Congress on Engineering*. Vol. 2008. 2008.

[13] Alessandro Abate et al. "Formal synthesis of Lyapunov neural networks". In: *IEEE Control Systems Letters* 5.3 (2020), pp. 773–778.

[14] Hongkai Dai et al. "Counter-example guided synthesis of neural network Lyapunov functions for piecewise linear systems". In: *2020 59th IEEE Conference on Decision and Control (CDC).* 2020, pp. 1274–1281. DOI: `10.1109/CDC42340.2020.9304201`.

[15] Hongkai Dai et al. *Lyapunov-stable neural-network control.* 2021. DOI: 10.48550/ARXIV.2109.14152. URL: https://arxiv.org/abs/2109.14152.

[16] Nathan Gaby, Fumin Zhang, and Xiaojing Ye. "Lyapunov-Net: A Deep Neural Network Architecture for Lyapunov Function Approximation". In: *CoRR* abs/2109.13359 (2021). arXiv: 2109.13359. URL: https://arxiv.org/abs/2109.13359.

[17] Lars Grüne. *Computing Lyapunov functions using deep neural networks.* 2020. DOI: 10.48550/ARXIV.2005.08965. URL: https://arxiv.org/abs/2005.08965.

[18] William La Cava et al. *Contemporary Symbolic Regression Methods and their Relative Performance.* 2021. DOI: 10.48550/ARXIV.2107.14351. URL: https://arxiv.org/abs/2107.14351.

[19] John R Koza. "Genetic programming as a means for programming computers by natural selection". In: *Statistics and computing* 4.2 (1994), pp. 87–112.

[20] Gregory S Hornby. "ALPS: the age-layered population structure for reducing the problem of premature convergence". In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation.* 2006, pp. 815–822.

[21] Michael D Schmidt and Hod Lipson. "Age-fitness pareto optimization". In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation.* 2010, pp. 543–544.

[22] William La Cava et al. "A probabilistic and multi-objective analysis of lexicase selection and ε-lexicase selection". In: *Evolutionary Computation* 27.3 (2019), pp. 377–402.

[23] Ying Jin et al. "Bayesian symbolic regression". In: *arXiv preprint arXiv:1910.08892* (2019).

[24] Silviu-Marian Udrescu and Max Tegmark. "AI Feynman: A physics-inspired method for symbolic regression". In: *Science Advances* 6.16 (2020), eaay2631. DOI: 10.1126/sciadv.aay2631. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.aay2631. URL: https://www.science.org/doi/abs/10.1126/sciadv.aay2631.

[25] Silviu-Marian Udrescu et al. "AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 4860–4871.

[26] Benyamin Grosman and Daniel R. Lewin. "Lyapunov-based stability analysis automated by genetic programming". In: *Automatica* 45.1 (2009), pp. 252–256. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2008.07.014. URL: https://www.sciencedirect.com/science/article/pii/S0005109808004159.

[27] Jeff S McGough, Alan W Christianson, and Randy C Hoover. "Symbolic computation of Lyapunov functions using evolutionary algorithms". In: *Proceedings of the 12th IASTED international conference.* Vol. 15. 2010, p. 17.

[28] Miles Cranmer. *PySR: Fast & Parallelized Symbolic Regression in Python/Julia.* Sept. 2020. DOI: 10.5281/zenodo.4041459. URL: http://doi.org/10.5281/zenodo.4041459.

[29] M. Hohenwarter et al. *GeoGebra 4.4.* http://www.geogebra.org. Dec. 2013.