MATH181 Project: Automatic Identification of Nonlinear Systems

Theo Rode

1 Introduction

This project discusses the procedure of recovering the underlying equations for the dynamics given noisy data collected by the system. So called "physics-inspired" or "physics-informed" AI. The goal of this field is to gain a physical understanding of systems where we can measure various trajectories, but the dynamics are unknown. Though this is additionally useful in situations where complex systems have dynamical properties that can be modeled on a smaller scale. In particular, the focus of this project is on discussing the results demonstrated in a paper published in 2016 by Steven Brunton, Joshua Proctor, and Nathan Kutz who propose an algorithm for reconstructing dynamics dubbed SINDy (Sparse Identification of Nonlinear Dynamics) [4]. This first section aims to discuss the field more broadly and relevant papers to the results proposed in the aforementioned paper as well as more generally in the field.

The authors of [4] give two foundational papers for recovering the dynamics of nonlinear systems: a paper by Josh Bongard and Hod Lipson [3] and one by Michael Schmidt and Hod Lipson [14], in 2007 and 2009 respectively. Bongard and Lipson [3] outline a method for "reverse engineering" the dynamics of nonlinear systems utilizing an active learning system. The main process of the algorithm is a technique labeled **symbolic regression**. In [3], symbolic regression is made usable for complicated (nonlinear) systems by introducing three concepts: partitioning, automated probing, and snipping. Partitioning is a process by which the algorithm is able to model each of the variables in the system separately using Stochastic optimization. Each of the variables are integrated by substituting in representations of the other variables rather than integrating the variables together. This method significantly reduces the overhead when working with high dimensional systems. Automated probing allows the algorithm to explore the system through active learning. After candidate models are created, the algorithm attempts to create many "tests" which are judged against their ability to disprove as many of the candidate models as possible. Then, the test which rejects as many of the candidate models as possible is used to judge the performance of all candidate models in order to pick the most capable ones. From this, the next generation of candidate models can be generated. Finally, snipping is a process to reduce the complexity of the generated models and prevent over-fitting. In particular, when creating another generation of models, occasionally certain subexpressions will be replaced with a value of the subexpression picked uniformly from its range. This acts to reduce situations where the models will create complex subexpressions which take on narrow ranges of values, demonstrating over-fitting to the data. Together, these innovations on the process of symbolic regression allowed Bongard and Lipson to create a framework that was applicable to an array of real-world systems. The framework demonstrated the ability to find nonlinearities and interdependencies as well as an ability to scale successfully. However, the authors note that current limitations include not yet demonstrating scalability to much larger biological models and an inability to work on data where certain variables are unobservable, as a key component of the framework was the active learning [3].

Schmidt and Lipson present a different algorithm for reconstructing dynamics that is still based on **symbolic regression** [14]. This paper focuses on presenting a new metric for comparing the accuracy of various candidate equations in order to better search for invariants within the system. In particular, once candidate equations are generated (initially randomly from a set of "building blocks"), the partial derivative pairings of every pair of variables are compared to the numerically calculated partial derivative pairings in order to select the equations which model these pairs the closest. The next generation of equations is then generated from these best equations probabilistically. The algorithm will return a set of equations when they reach a certain accuracy to the observed data, calculated from a subset of the data randomly selected to be withheld from training. Furthermore, these equations are prioritized according to a measured parsimony, effectively complexity which is calculated by the number of terms in the expression. The authors were able to demonstrate that this algorithm could reconstruct invariants from numerous physical systems effectively. Additionally, they noticed that with a limited set of building blocks, the algorithm found approximations to functions like sines and cosines, utilizing the available building blocks, when such functions were part of the physical invariant [14].

2 Mathematical Background

For the SINDy algorithm, the authors aimed to reframe the problem of recovering nonlinear dynamics using **sparse regression** and **compressed sensing** [4]. Both sparse regression and compressed sensing broadly refer to the process of finding a regression solution to a problem such that the solution is sparse in some space [4]. Brunton et al. relate that utilizing sparse regression is a strategy to reduce the noise present in the fitting of systems [4]. The particular assumptions for sparse regression come from the idea that in most cases, the dynamics for a particular system will be governed by a small set of given basis functions. Therefore, the sparsity also gives advantages for computation as we can increase the size of the function space dramatically without needing significantly more computation [4].

The authors of [4] propose an iterative algorithm for sparse regression. In particular, they start with a least-squares fit to the data, and then iteratively remove coefficients below a given threshold. By refitting with the remaining, nonzero terms iteratively, the algorithm can arrive at a sparse solution [4].

The SINDy algorithm relies on the availability of both sampled data, $\mathbf{X}$, and the sampled derivative, $\dot{\mathbf{X}}$. However, the authors note that the availability of derivative measurements are not guaranteed and therefore propose using the **total variation regularized derivative** to mitigate the noise created from numerically calculating $\dot{\mathbf{X}}$ from $\mathbf{X}$ [4]. In particular, they cite the algorithm used by Rudin et al. [13]. Here, Rudin et al. use a framework for computing a desired, denoised output, u, from a noisy input, u_0, by assuming that $u = u_0 + n$ for some white noise function n. It is assumed that n has a mean of 0 with some given standard deviation. The minmization problem corresponds to minimizing $\int_\Omega \sqrt{u_x^2 + u_y^2}\,\mathrm{d}A$, which is contrained by $\int_\Omega u\,\mathrm{d}A = \int_\Omega u_0\,\mathrm{d}A$ under the assumption of the additive white noise. It is then proposed to solve for u through either an iterative algorithm (i.e. simulated annealing) or, preferably, a PDE solver to find local minimum [13].

2.1 SINDy

The most basic systems considered for the algorithm SINDy are those of the form

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)).$$

In order to use the concepts of sparse regression and compressed sensing, discussed above, the authors assume that $\mathbf{f}$ is sparse in some function space [4]. The state vector, $\mathbf{x}(t)$, is collected over some time period and $\dot{\mathbf{x}}$ is either sampled alongside $\mathbf{x}(t)$ or numerically calculated from it. These samplings create the following matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix},$$

and the immediately following $\dot{\mathbf{X}}$. We can then construct a library of candidate functions:

$$\Theta(\mathbf{X}) = \begin{bmatrix} | & | & | & | & & | & | & \\ 1 & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \cdots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \cdots \\ | & | & | & | & & | & | & \end{bmatrix}.$$

Here, $\mathbf{X}^{P_2}$ represents the quadratic nonlinearities in $\mathbf{f}$, in particular:

$$\mathbf{X}^{P_2} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1)x_2(t_1) & \cdots & x_1(t_1)x_n(t_1) & x_2^2(t_1) & \cdots & x_n^2(t_1) \\ x_1^2(t_2) & x_1(t_2)x_2(t_2) & \cdots & x_1(t_2)x_n(t_2) & x_2^2(t_2) & \cdots & x_n^2(t_2) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_1^2(t_m) & x_1(t_m)x_2(t_m) & \cdots & x_1(t_m)x_n(t_m) & x_2^2(t_m) & \cdots & x_n^2(t_m) \end{bmatrix}.$$

Similarly, $\sin(\mathbf{X})$ would be

$$\sin(\mathbf{X}) = \begin{bmatrix} \sin(x_1(t_1)) & \sin(x_2(t_1)) & \cdots & \sin(x_n(t_1)) \\ \sin(x_1(t_2)) & \sin(x_2(t_2)) & \cdots & \sin(x_n(t_2)) \\ \vdots & \vdots & \ddots & \vdots \\ \sin(x_1(t_m)) & \sin(x_2(t_m)) & \cdots & \sin(x_n(t_m)) \end{bmatrix}.$$

Therefore, each column of $\Theta(\mathbf{X})$ represents a candidate function for $\mathbf{f}$. From this, we set up the sparse regression problem

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\boldsymbol{\Xi}.$$

In particular, $\boldsymbol{\Xi} = \begin{bmatrix} \boldsymbol{\xi}_1 & \boldsymbol{\xi}_2 & \cdots & \boldsymbol{\xi}_n \end{bmatrix}$ where each $\boldsymbol{\xi}_i$ is a sparse vector dictating which functions in the function library are active for each of the row equations in $\mathbf{f}$. The sparse regression problem is an iterative algorithm that iteratively thresholds out coefficients in $\boldsymbol{\Xi}$ under some sparsity parameter, λ, and then refits the problem on the remaining functions.

The authors additionally suggest a slight modification to the algorithm in order to detect bifurcations present in a system. In particular, the authors appended the bifurcation parameter to the dynamics:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}; \mu), \\ \dot{\mu} &= 0. \end{aligned} \tag{1}$$

Then, by sampling data from different values of μ, it is possible for SINDy to reconstruct the system from combinations of $\mathbf{x}$ as well as μ. In turn, the system can then find the dynamics present with changes in μ, and hence bifurcations.

3 Results

3.1 Code

The main object in recreating the results presenting in the paper was writing a codebase that is able to recreate the SINDy algorithm. I wrote the code in the Julia Programming Language [2] and with all plotting done in PlotlyBase.jl [11].

All data used to train the SINDy model was collected by integrating using ordinary differential equation (ODE) solvers provided by the DifferentialEquations.jl [12] package. For the numerical stability of the model, data was collected at fixed timesteps, as shown in Listing 1 for the Lorenz system.

Listing 1: Lorenz system data simulation

```julia
function lorenz_data(u0, tspan, p, dt)
    system = ODEProblem(lorenz!, u0, tspan, p) # define ODE problem
    times = collect((tspan[1]):dt:(tspan[2])) # constant timestamps

    solve(system, DP5(), dt = dt, saveat = times) # solve with Explicit Runge-
        Kutta Method.
end # function lorenz_data
```

Listing 2: Data and derivative library construction

```
function constructX(data::Vector{Vector{Float64}}, tvdiff_params::TvDiffParams,
     t::Vector{Float64})
    X::Matrix{Float64} = vcat(transpose.(data)...)

    variable_derivatives = [] # derivatives

    # differentiate each variable separately
    for n in 1:(size(X)[2])
        # get interpolation
        itp = linear_interpolation(t, X[:, n])

        if tvdiff_params.perform_tvdiff
            # assume it was computed with equal dt
            dt = t[2] - t[1]

            push!(variable_derivatives,
                tvdiff(X[:, n], tvdiff_params.iterations, tvdiff_params.α, ε =
                     tvdiff_params.ε, scale = tvdiff_params.scale, precond =
                     tvdiff_params.precond, dx = dt, diff_kernel = tvdiff_params
                     .diff_kernel)
            )
        else
            # get derivatives
            grad::Vector{Float64} = [Interpolations.gradient(itp, ti)[1] for ti
                 in t]
            push!(variable_derivatives, grad)
        end # if tvdiff_params.perform_tvdiff
    end # for n

    Xdot::Matrix{Float64} = hcat(variable_derivatives...)

    (X, Xdot)
end # function constructX
```

Additionally, noise sampled from a standard normal distribution was scaled by a noise parameter and added to the collected data before SINDy was trained on it. This code utilized the Distributions.jl [1][9] package to provide the normal distribution. Once the data was collected and noise applied, the program constructed the data matrix, $\mathbf{X}$, and the corresponding derivative matrix, $\dot{\mathbf{X}}$. Differentiation is done in one of two ways, either through direct numerical differentiation, using Interpolations.jl [8], or using the total-variation regularization derivative, provided by NoiseRobustDifferentiation.jl [5][7]. Using a linear interpolation to compute the derivatives had the additional benefit of being more stable when samples were not taken with constant timestamps, and deviates in approach from the code written by Brunton et al. [4]. The code for constructing the data and derivative libraries is shown in Listing 2.

Utilizing the data matrix, $\mathbf{X}$, it is then simple to construct the function library, $\mathbf{\Theta}(\mathbf{X})$. Unlike the original paper [4], this implementation allows for arbitrary functions to be used and provides a framework for easily expanding the type and quantity of functions used. In particular, along with the potential to fit arbitrary functions, the generation of polynomial terms was done generally

Listing 3: Sparse regression

```
function sparse_regression(Θ::Matrix{Float64}, Xdot::Matrix{Float64}, λ::Float64
    , kmax::Integer)
    # create initial guess for Xi as just regression
    Ξ = Θ\Xdot

    for _ in 1:kmax
        # find coefficients that are less than λ
        small_coefficients = abs.(Ξ) .< λ

        # set small coefficients to 0
        Ξ[small_coefficients] .= 0

        # now individually run regression on non-zero coefficients
        for n in 1:(size(Xdot)[2]) # loop through all state dimensions
            nonsmall = .!small_coefficients[:, n]

            Ξ[nonsmall, n] = Θ[:, nonsmall]\Xdot[:, n]
        end # for n
    end # for k

    Ξ
end # function sparse_regression
```

to support arbitrary degree polynomials utilizing functions Combinatorics.jl [6].

Finally, the sparse regression algorithm to compute $\mathbf{\Xi}$ is the same process as outlined by Brunton et al. [4]. In particular, the starting point for the iterative algorithm is a least-squares solution to the problem using a standard regression. Then, over k iterations, coefficients in $\mathbf{\Xi}$ which are less than the sparsity parameter, λ, are dropped and replaced with zero. In each of these iterations, the regression is then recomputed with only the nonzero entries. This code is presented in Listing 3.

The resulting system defined by $\mathbf{\Xi}$ can then be solved using the same methods as when collecting data for the original system. The governing equations are constructed using the same function generation from the library construction in order to ensure complete generality.

3.2 The Lorenz Attractor

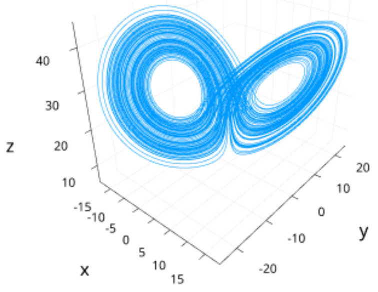
In replicating the results of Brunton et al., I first pursued replicating the success they found in recoverying the dynamics present in the chaotic Lorenz System [4][10]. The dynamics for this system are governed by the following equations,

$$\begin{aligned}\dot{x} &= \sigma(y - x),\\ \dot{y} &= x(\rho - z) - y,\\ \dot{z} &= xy - \beta z.\end{aligned} \tag{2}$$

I used the same initial conditions and parameters that Brunton et al. [4] used for their demonstrations. In particular, $u_0 = (-8, 7, 27)$ and parameters $(\sigma, \rho, \beta) =$

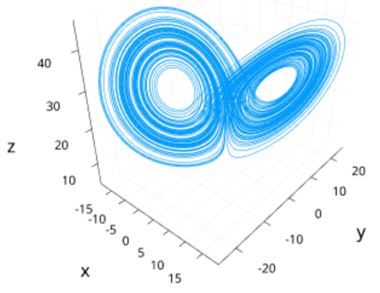
$\left(10, 28, \frac{8}{3}\right)$. Figure 1 shows the actual system with these parameters and initial condition.

Fig. 1: Lorenz system

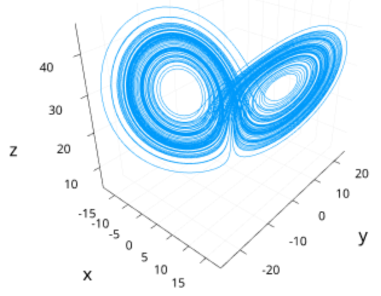


I ran the SINDy algorithm on both data with noise of magnitude $\eta = 0.01$ and $\eta = 0.1$. Of interest is the system with $\eta = 0.1$ was only able to converge on a solution when the derivative was computed using the total-variation regularization derivative. For a function basis, I chose a basis comprised of all 7th-degree polynomial terms and all sines and cosines of the form $\sin(ku_i)$ and $\cos(ku_i)$ for all $\{u_i\} = \{x, y, z\}$ and integer k from 1 to 10. This created a basis of 180 possible functions. Further, the sparsification parameter was chosen to be $\lambda = 0.25$ with 10 iterations for sparse regression. Integration of the input data was done over a timespan from 0 to 100 with $dt = 0.001$. The resulting systems from $\eta = 0.01$ are shown in Figure 2 and the system resulting from $\eta = 0.1$ is shown in Figure 3.

Fig. 2: Discovered Lorenz system with noise $\eta = 0.01$



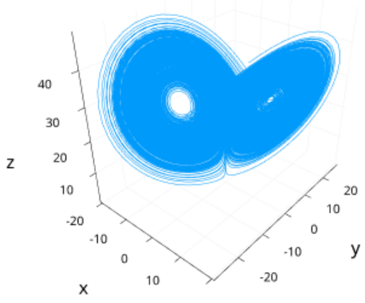
With standard derivative calculation



With total-variation regularization derivative

We see that the SINDy algorithm was able to effectively capture the attracting dynamics present in the Lorenz system in all cases. Furthermore, for the

Fig. 3: Discovered Lorenz system with noise $\eta = 0.1$ and total-variation regularization differentiation



standard derivative and noise $\eta = 0.01$, the discovered system had the form

$$\begin{aligned} \dot{x} &= -9.982573809940325x + 9.999536628579511y, \\ \dot{y} &= 27.573169299598813x - 0.8370109395705518y - 0.98935201514641xz, \\ \dot{z} &= -2.660754272317803z + 0.9988807483584906xy. \end{aligned} \tag{3}$$

Comparing this result to the actual system (Equation 2), we see that the discovered system has perfectly identified the polynomial terms in the system with coefficients that are almost identical. We see similar strength in identifying the system for $\eta = 0.01$ (Equation 4) and $\eta = 0.1$ (Equation 5) systems with total-variation regularization derivatives.

$$\begin{aligned} \dot{x} &= -9.946417397865872x + 9.97621578174998y, \\ \dot{y} &= 27.0485300793758x - 0.670689440675418y - 0.976667757332429xz, \\ \dot{z} &= -2.6495215942228163z + 0.996381152512881xy. \end{aligned} \tag{4}$$

$$\begin{aligned} \dot{x} &= -9.805048724434341x + 9.835722669096466y, \\ \dot{y} &= 25.92133086771059x - 0.433587521465246y - 0.946115512156319xz, \\ \dot{z} &= -1.930341755171950 - 2.54324607967731z + 0.983562354075398xy. \end{aligned} \tag{5}$$

We do see a forcing term introduced in the higher noise, $\eta = 0.1$, system for $\dot{z}$ which represents a term not present in actual system or the other two discovered systems. Though decreased performance is expected with higher noise.

3.3 Bifurcations: The Hopf normal form

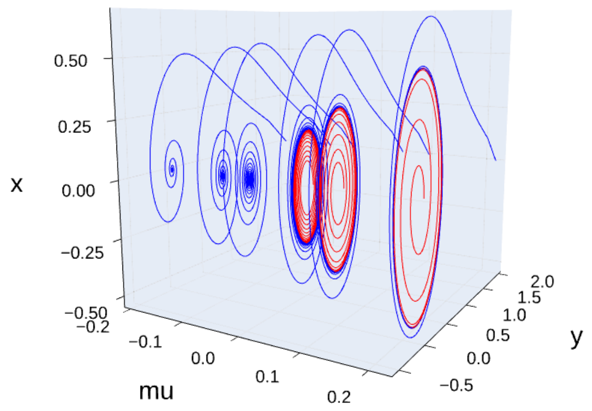
In addition to SINDy's ability to extract the system dynamics from the chaotic Lorenz attractor (3.2), Section 2.1 discussed how Brunton et al. discussed a modification to SINDy which would allow the detection of bifurcations in a system [4]. Similar to the authors, I will demonstrate this using the Hopf normal form:

$$\begin{aligned} \dot{x} &= \mu x - \omega y - Ax(x^2 + y^2), \\ \dot{y} &= \omega x + \mu y - Ay(x^2 + y^2). \end{aligned} \tag{6}$$

While μ is the bifurcation parameter and will be varried, I will take $\omega = A = 1$. In order for SINDy to discover the bifurcation in its system, we have to collect data over a range of μ values. In particular, I collected data for 16 values of μ between -0.2 and 0.55, with a Hopf bifurcation occurring when $\mu = 0$. For negative values of μ, I collected data with the initial condition $(x_0, y_0) = (2, 0)$, while for positive values data was collected both for this initial condition and for $(x_0, y_0) = (0.01, 0)$ in order to capture the limit cycle behavior. Further, all collected data was given noise of magnitude $\eta = 0.005$.

In order to account for multiple orbits worth of data, the derivatives for each orbit were calculated separately using the total-variation regularization derivative. Then, the $\mathbf{X}$ and $\dot{\mathbf{X}}$ matrices for all orbits were combined into a single data and derivative matrix. These could then be passed through SINDy as previously. For this example, a basis of all polynomials through 5th degree was used, with sparsity parameter $\lambda = 0.25$ and 10 iterations of the sparse regression. Figure 4 shows the discovered system as various values of μ and its ability to detect the Hopf bifurcation.

Fig. 4: Discovered Hopf bifurcation. Blue orbits start at $(x_0, y_0) = (2, 0)$ and red orbits start at $(x_0, y_0) = (0.01, 0)$.



Furthermore, SINDy found the system to be given by

$$\begin{aligned} \dot{x} &= -0.9916y + 0.9314\mu x - 0.9178x^3 - 0.9178xy^2 - 0.3081x^4y, \\ \dot{y} &= 0.9972x + 0.9926\mu y - 0.9811x^2y - 0.9814y^3. \end{aligned} \tag{7}$$

Here we notice that barring the $-0.3081x^4y$ term in the equation for $\dot{x}$, the algorithm was able to identify the constants remarkably well and the nonlinear terms accurately. Though, its important to note that depending on the noise added to the system, the resulting system can be significantly worse. In particular, Equation 8 resulted from the same parameters but with different noise applied.

$$\begin{aligned}\dot{x} &= -0.9971y + 0.8704\mu x - 0.8617x^3 - 0.9177xy^2 + 0.9186\mu x^2 y - 0.2853xy^3 \\ &\quad - 0.9918\mu xy^2 - 0.8546x^4 y + 1.2614x^3 y^2 - 1.0363x^2 y^3 + 1.0065xy^4 \\ &\quad + 0.5446\mu y^3 x, \\ \dot{y} &= 0.9968x + 0.9930\mu y - 0.9807x^2 y - 0.9822y^3.\end{aligned} \tag{8}$$

Interestingly, while $\dot{y}$ remained stable, $\dot{x}$ tends to me less stable. It is important to note, that while there are many additional higher-order polynomial terms, the qualatative behavior of the system is identical, still finding a Hopf bifurcation.

References

[1] Mathieu Besançon et al. “Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem”. In: *Journal of Statistical Software* 98.16 (2021), pp. 1–30. ISSN: 1548-7660. DOI: 10.18637/jss.v098.i16. URL: https://www.jstatsoft.org/v098/i16.

[2] Jeff Bezanson et al. “Julia: A fresh approach to numerical computing”. In: *SIAM review* 59.1 (2017), pp. 65–98. URL: https://doi.org/10.1137/141000671.

[3] Josh Bongard and Hod Lipson. “Automated reverse engineering of nonlinear dynamical systems”. In: *Proceedings of the National Academy of Sciences* 104.24 (2007), pp. 9943–9948.

[4] Steven L. Bruton, Joshua L. Proctor, and J. Nathan Kutz. “Discovering governing equations from data by sparse identification of nonlinear dynamical systems”. In: *Proceedings of the national academy of sciences* 113.15 (2016), pp. 3932–3937. DOI: https://doi.org/10.1073/pnas.1517384113.

[5] Rick Chartrand. “Numerical differentiation of noisy, nonsmooth data”. In: *International Scholarly Research Notices* 2011 (2011).

[6] Jiahao Chen et al. *Combinatorics.jl.* https://github.com/JuliaMath/Combinatorics.jl.

[7] Adrian Hill, Sathvik Bhagavan, and A. Ziehe. *NoiseRobustDifferentiation.jl.* https://github.com/adrhill/NoiseRobustDifferentiation.jl. Version v0.2.4.

[8] Mark Kittisopikul, Timothy E. Holy, and Tomas Aschan. *JuliaMath/Interpolations.jl: v0.14.7.* https://github.com/JuliaMath/Interpolations.jl. Version v0.14.7. Dec. 10, 2022.

[9] Dahua Lin et al. *JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions.* July 2019. DOI: 10.5281/zenodo.2647458. URL: https://doi.org/10.5281/zenodo.2647458.

[10] Edward N Lorenz. “Deterministic nonperiodic flow”. In: *Journal of atmospheric sciences* 20.2 (1963), pp. 130–141.

[11] Spencer Lyon and et al. *PlotlyBase.jl.* https://github.com/sglyon/PlotlyBase.jl. Version v0.8.18. Sept. 21, 2021.

[12] Christopher Rackauckas and Qing Nie. “DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia”. In: *Journal of Open Research Software* 5.1 (2017).

[13] Leonid Rudin, Stanley Osher, and Emad Fatemi. “Nonlinear total variation based noise removal algorithms.” In: *Physica D: nonlinear phenomena* 60 (1992), pp. 259–268.

[14] Michael Schmidt and Hod Lipson. “Distilling free-form natural laws from experimental data”. In: *science* 324.5923 (2009), pp. 81–85.