Paper

Computation of bifurcations: Automatic provisioning of variational equations

Seiya Amoh [1a)], *Miho Ogura* [1], *and Tetsushi Ueta* [1]

[1] *Graduate School of Advanced Technology and Science, Tokushima University, 2–1 Minami Josanjima, Tokushima, Tokushima, Japan*

[a)] *c502147001@tokushima-u.ac.jp*

Received October 18, 2021; Revised December 18, 2021; Published April 1, 2022

Abstract: In the conventional implementations for solving bifurcation problems, Jacobian matrix and its partial derivatives regarding the given problem should be provided manually. This process is not so easy, thus it often induces human errors like computation failures, typing error, especially if the system is higher order. In this paper, we develop a preprocessor that gives Jacobian matrix and partial derivatives symbolically by using SymPy packages on the Python platform. Possibilities about the inclusion of errors are minimized by symbolic derivations and reducing loop structures. It imposes a user only on putting an expression of the equation into a JSON format file. We demonstrate bifurcation calculations for discrete neuron dynamical systems. The system includes an exponential function, which makes the calculation of derivatives complicated, but we show that it can be implemented simply by using symbolic differentiation.

Key Words: chaos, bifurcation, symbolic differentiation

1. Introduction

Fixed points occurring in discrete dynamical systems often cause bifurcation phenomena due to parameter changes. These changes in topological properties due to bifurcations can provide important information for understanding the property of dynamical systems, but it is somewhat difficult to calculate the bifurcation set. In the conventional method[1], Jacobian matrix and its partial derivative for the system must be calculated and programmed manually. This is a major cause of manual calculation errors and human errors in programming, especially for high-dimensional systems. Also, there is a method that uses numerical differentiation to perform numerical calculations, but in this case, numerical errors are clearly present and the convergence accuracy is poor. These problems are the same for bifurcation analysis packages that have been developed in the past, AUTO[2] for example.

In recent years, the performance of interpreted languages has improved greatly, and they provide a very useful platform for scientific computing libraries that are essential for nonlinear dynamical systems. In this paper, we propose to simplify the construction of variational equations, which is considered to be the most difficult part of creating bifurcation calculations, by describing them using symbolic differentiation. This bifurcation calculation packages using this method only require the user to define a map. In other words, there is no need to prepare complicated variational equations in advance. In addition, this method uses an interpreted language instead of the conventional compiled language. This makes the size of the bifurcation computation package more compact.

Bifurcation phenomena in discrete neuron dynamical systems have been conventionally studied to improve the learning efficiency of neural networks. Recently, new activation functions have been proposed to improve the accuracy of learning in neural networks. In this study, as an example of applying the proposed method, we analyze the bifurcation of a system using the Swish function, which has high nonlinearity, as the activation function. Thus, this system is difficult to calculate the partial derivatives by hand due to the activation function. However, symbolic differentiation of the proposed method makes it easy to check the bifurcation structure.

2. Symbolic Differentiation of Discrete Systems

Consider a discrete system written as $\boldsymbol{x}_{k+1} = T_\lambda(\boldsymbol{x}_k)$, where, $\boldsymbol{x} \in \boldsymbol{R}^n$, $T_\lambda : \boldsymbol{R}^n \mapsto \boldsymbol{R}^n$ and T is differentiable. The bifurcation condition for the ℓ-periodic point is

$$\begin{cases} T_\lambda^\ell(\boldsymbol{x}_0) - \boldsymbol{x}_0 = \boldsymbol{0} \\ \chi(\boldsymbol{x}_0, \lambda) = \det\left(\dfrac{\partial T_\lambda^\ell}{\partial \boldsymbol{x}}(\boldsymbol{x}_0) - \mu I\right) = 0, \end{cases} \tag{1}$$

where a parameter $\lambda \in \boldsymbol{R}$ and a characteristic constant $\mu \in \boldsymbol{C}$. T_λ^ℓ denotes that the map T_λ is applied ℓ times. We set μ to 1 or -1 for each of tangent bifurcation and period-doubling bifurcation. When calculating the Nemark-Sacker bifurcation, we solve the Bialternate product condition[3] or $n+2$ simultaneous equations by dividing the second equation in Eq. (1) into real and imaginary parts[4]. To compute the bifurcation set, we can solve Eq.(1) for $(\boldsymbol{x}_0, \lambda)$ using the Newton's method, which requires partial differentiation of the objective function by the state variables and the parameter. In order to obtain these partial derivatives $\partial T_\lambda^\ell/\partial \boldsymbol{x}_0$, $\partial T_\lambda^\ell/\partial \lambda$, $\partial \chi/\partial \boldsymbol{x}_0$, and $\partial \chi/\partial \lambda$, we use SymPy package of Python platform to perform symbolic differentiation.

Listing 1 shows an example of coding. Note that the second partial derivatives of T_λ are needed to calculate the partial derivative of χ. In SymPy, algebraic structures are treated as objects, so once the derivative objects are computed, the values of the variational equations can be obtained by simply substituting state values and parameter values to them.

Listing 1: An example of symbolic differentiation

```
# xdim is the dimension of the system,
# pdim is the number of parameters,
# and var_param is variable parameter index.
x = sympy.MatrixSymbol('x', xdim, 1)
p = sympy.MatrixSymbol('p', pdim, 1)
T = map(x, p)
dTdx = T.jacobian(x)
dTdlambda = sympy.diff(T, p[var_param])
for i in range(xdim):
        dTdxdx[i] = sympy.diff(dTdx, x[i])
dTdxdlambda = sympy.diff(dTdx, p[var_param])
```

`jacobian(<VECTOR_SYMBOL>)` is literally a method for partial differentiation of a function according to a vector variable `<VECTOR_SYMBOL>`. This makes it easy to derive $\partial T_\lambda/\partial \boldsymbol{x}$. `diff(<FUNCTION>, <SYMBOL>)` is a method to differentiate `<FUNCTINO>` by a scalar variable `<SYMBOL>`. By doing so, we can obtain the second-order partial derivative from $\partial T_\lambda/\partial \boldsymbol{x}$. To substitute values, we use `subs(...)` method. For example, we obtain the array of the first derivative of T^l by giving a Python list of taples of symbolic variables and value arrays as `dTdx_value = dTdx.subs([(x, x_0), (p, p_0)])`, where, `x_0` and `p_0` are values of the fixed point and parameters.

The derivative of the determinant is not differentiated symbolically in this method, so it must be written manually. Using multilinearity of the determinant, the function can be written as Listing 2. The arguments `A` and `dA` are $\partial T_\lambda^\ell/\partial \boldsymbol{x}_0 - \mu I$ and its derivative, respectively.

Listing 2: An example implementation of determinant differentiation

```python
def det_derivative(A, dA):
    ret = 0+0j
    for i in range(xdim):
        temp = A.copy()
        temp[:, i] = dA[:, i]
        ret += np.linalg.det(temp)
    return ret
```

The method of obtaining derivatives completely by symbolic differentiation shown in Listing 1 a very tractable design for systems with simple structures. However, when the map T_λ is defined by a complex structure such as a hyperbolic function, the output of derivatives by symbolic differentiation become complicated and numerical errors are introduced. This problem is especially apparent when computing multi-period point if we describe `T = map(map(...))` in Listing 1. In order to solve this problem, it is necessary to write the chain rule for the derivative of the composite map to the program.

Derivatives of the composite map T_λ^ℓ are derived as:

$$\frac{\partial T_\lambda^\ell}{\partial \boldsymbol{x}} = \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-2}} \cdots \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{1}} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{0}}, \tag{2}$$

$$\begin{aligned}
\frac{\partial T_\lambda^\ell}{\partial \lambda} &= \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}} \frac{\partial T_\lambda^{\ell-1}}{\partial \lambda} + \left.\frac{\partial T_\lambda}{\partial \lambda}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}}, \\
\frac{\partial T_\lambda^j}{\partial \lambda} &= \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1}} \frac{\partial T_\lambda^{j-1}}{\partial \lambda} + \left.\frac{\partial T_\lambda}{\partial \lambda}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1}}, \\
\frac{\partial T_\lambda^1}{\partial \lambda} &= \left.\frac{\partial T_\lambda}{\partial \lambda}\right|_{\boldsymbol{x}=\boldsymbol{x}_{0}},
\end{aligned} \tag{3}$$

$$\begin{aligned}
\frac{\partial^2 T_\lambda^\ell}{\partial \boldsymbol{x}^2} = {} & \left.\frac{\partial^2 T_\lambda}{\partial \boldsymbol{x}^2}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}} \prod_{k=0}^{\ell-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-2-k}} \prod_{k=0}^{\ell-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-2-k}} \\
& + \cdots + \prod_{k=0}^{j+1} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1-k}} \left.\frac{\partial^2 T_\lambda}{\partial \boldsymbol{x}^2}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j}} \prod_{k=0}^{j-1} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1-k}} \prod_{k=0}^{j-1} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1-k}} \\
& + \cdots + \prod_{k=0}^{\ell-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1-k}} \left.\frac{\partial^2 T_\lambda}{\partial \boldsymbol{x}^2}\right|_{\boldsymbol{x}=\boldsymbol{x}_{0}},
\end{aligned} \tag{4}$$

$$\begin{aligned}
\frac{\partial T_\lambda^\ell}{\partial \boldsymbol{x}\partial\lambda} &= \left.\frac{\partial^2 T_\lambda}{\partial \boldsymbol{x}^2}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}} \prod_{k=0}^{\ell-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-2-k}} \frac{\partial T_\lambda^{\ell-1}}{\partial \lambda} + \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}} \frac{\partial T_\lambda^{\ell-1}}{\partial \boldsymbol{x}\partial\lambda} + \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}\partial\lambda}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-1}} \prod_{k=0}^{\ell-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{\ell-2-k}}, \\
\frac{\partial T_\lambda^j}{\partial \boldsymbol{x}\partial\lambda} &= \left.\frac{\partial^2 T_\lambda}{\partial \boldsymbol{x}^2}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1}} \prod_{k=0}^{j-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-2-k}} \frac{\partial T_\lambda^{j-1}}{\partial \lambda} + \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1}} \frac{\partial T_\lambda^{j-1}}{\partial \boldsymbol{x}\partial\lambda} + \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}\partial\lambda}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-1}} \prod_{k=0}^{j-2} \left.\frac{\partial T_\lambda}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{x}_{j-2-k}}, \\
\frac{\partial T_\lambda^1}{\partial \boldsymbol{x}\partial\lambda} &= \left.\frac{\partial^2 T_\lambda}{\partial \boldsymbol{x}\partial\lambda}\right|_{\boldsymbol{x}=\boldsymbol{x}_{0}},
\end{aligned} \tag{5}$$

where, $j = \ell-1, \ell-2, \ldots, 3, 2$, $\boldsymbol{x}_k = T_\lambda^k(\boldsymbol{x}_0)$. Since the derivative of the composite map is constructed based on the concept of Ref.[5], bifurcation analysis can be performed for switching systems by appropriately defining the map piecewise.

In Eqs. (4) and (5), the second-order partial derivative which is "3-dimensional matrix"[6] appears, and since Python's `@` operator can perform the matrix contraction product of this 3-dimensional array, the equations can be programmed directly. For example, a product of $n \times n \times n$ array `A` and $n \times n$ array `B` can be written as `A@B`.

Note that Eqs. (3) and (5) has nest structure for $\partial T_\lambda^j/\partial\lambda$ and $\partial^2 T_\lambda^j/\partial\boldsymbol{x}\partial\lambda$, then the recursive implementation helps to keep the program simple like Listing 3. `dTdxk[k]` and `dTdlambdak[k]` are substitutions of state $\boldsymbol{x}_k$ for $\partial T_\lambda/\partial\boldsymbol{x}$ and $\partial T_\lambda/\partial\lambda$, respectively.

Listing 3: An example implementation of parameter derivatives

```
def dTldlambda():
    ret = dTdlambdak[0]
    for i in range(1, period):
        ret = dTdxk[i] @ ret + dTdlambdak[i]
    return ret
```

Manual coding of the composite map and its derivatives increase the programming cost, but provides better performance compared to the all symbolic differentiation method. Furthermore, this implementation also requires only the definition of the map T_λ, which does not change the availability of the package.

Figure 1 shows the flowchart of the bifurcation calculation program. First of all, prepare a JSON format file as an interface for the program. Since all information such as the definition of maps and initial values for bifurcation calculations is provided via the JSON file, high usability is guaranteed. A derivative object is generated using SymPy based on the definition of the given map. This operation is performed only once at program startup, and can be used in subsequent routines. The main routine is divided into three operations: assign values to the derivatives, compute the objective function and its Jacobian matrix values, and perform the Newton's method. The routine terminates when the number of parameter increments reaches the specified number, or when the Newton's method oscillates or diverges. δ, η, and ϵ in the figure represent the update difference of the Newton's method, the threshold for divergence, and the convergence tolerance, respectively. As mentioned before, manual coding of the derivative of the composite map is required at the step of the assignment operation to the derivatives to improve computational accuracy and performance. The manually coded program is used in the all following simulations of bifurcation calculations.

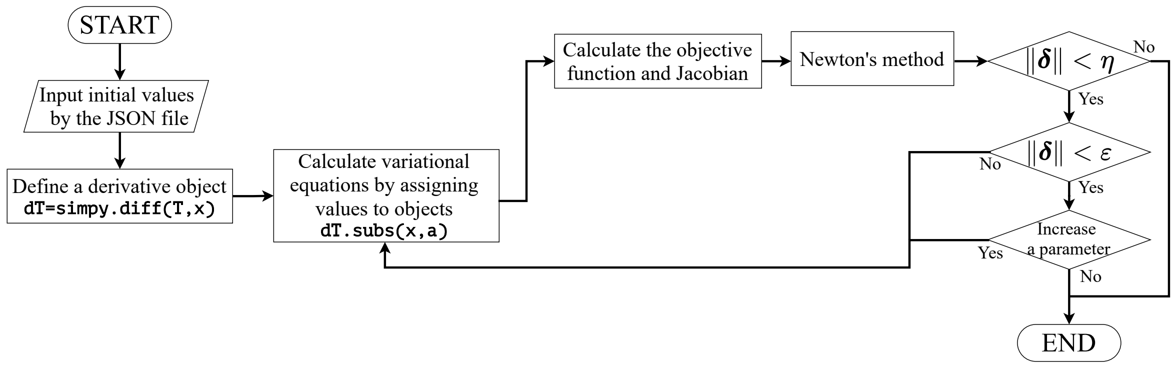

START
Input initial values by the JSON file
Define a derivative object dT=simpy.diff(T,x)
Calculate variational equations by assigning values to objects dT.subs(x,a)
Calculate the objective function and Jacobian
Newton's method
$\|\boldsymbol{\delta}\| < \eta$
No
Yes
$\|\boldsymbol{\delta}\| < \varepsilon$
No
Yes
Increase a parameter
Yes
No
END

Fig. 1: Flowchart of the bifurcation calculation program.

Listing 4 shows a JSON input design example. The JSON file contains a list of fixed points, parameters, etc., which are the initial values for bifurcation calculations. It can also be designed so that the definition of the map is performed in the JSON file. In this case, the definition of the map described in the input is executed using the built-in function `eval()`.

Listing 4: An example of a input JSON file

```
{"map": [ <DEFINITION OF THE MAP> ],
 "x0": [ <FIXED POINT COORDINATE> ],
 "params": [ <PARAMETER> ],
 "period": <PERIOD NUMBER>,
 "sigma": <REAL PART OF CHARACTERISTIC CONSTANT>,
 "omega": <IMAGINARY PART OF CHARACTERISTIC CONSTANT>,
 "inc_param": <INDEX NUMBER OF INCREMENTAL PARAMETER>,
 "var_param": <INDEX NUMBER OF VARIABLE PARAMETER>}
```

3. An Example

In recent years, neuron dynamical systems have been actively studied, and there are not only continuous-time models but also discrete-time models. In neural networks for learning, discrete neuron dynamical systems are useful in terms of computational cost. The periodic points in these systems are involved in the learning performance of the network, and bifurcation may occur for these points when the parameters are dynamically updated in the learning process[7].

In the case of learning, performance improvement by changing the activation function has been done, and various models have been proposed instead of the conventional sigmoid function[8]. In neuron dynamical systems, rich bifurcation phenomena have been observed in the previous research. However, when the activation function is made richer or the scale of the network increases, the complexity of the bifurcation structure is expected to increase due to nonlinearity. In this chapter, we will investigate the bifurcation structure of the system using the Swish function, which is one of the well-considered activation function.

Consider a two-coupled neuron dynamical system[9]:

$$\begin{cases} x_{k+1} = f(w_{11}x_k + w_{12}y_k) \\ y_{k+1} = f(w_{21}x_k + w_{22}y_k) \end{cases}, \tag{6}$$

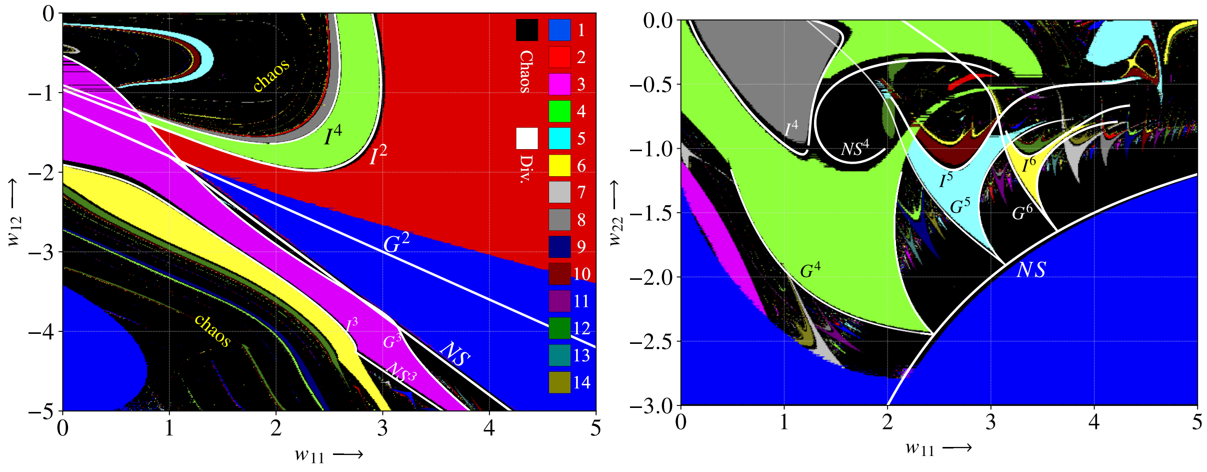
where w_{ij} are coupling coefficients. We use Swish function $f(z) = z/(1 + e^{-\xi z})$ for the activation function[10]. Since this system has an exponential in the denominator, the derivatives required for the Newton's method are complicated, but the bifurcation set can be calculated by using symbolic differentiation without hand calculation of derivatives.

By superimposing bifurcation curves computed by proposed algorithm and brute-force computation, we obtain Fig. 2a, where, $w_{21} = 5, w_{22} = -5, \xi = 1.5$. I^2 and G^2 show period-doubling and tangent bifurcations of the 2-periodic point, respectively. *NS* shows Neimark-Sacker bifurcation of the fixed point. There are chaotic regions at the top and bottom of the bifurcation diagram, and it is confirmed by the bifurcation set that the chaos is caused by period-doubling cascade. The boundary between the tangent bifurcation G^2 and 1 and 2 periodic points is misaligned because the brute-force algorithm tracks other fixed points, but the proposed method tracks the tangent bifurcation set exactly. In the 3-period region, the Neimark-Sacker bifurcation NS^3 is connected to the period-doubling bifurcation I^3, and chaos due to torus collapse occurs in this region. Figure 2b shows the bifurcation diagram at $w_{12} = -2, w_{21} = 5$. A 1-period region appears on the left side of the Neimark-Sacker bifurcation curve, where the Arnold tongue is predicted to exist. In fact, periodic and fixed points due to period locking coexist in the region. The 4-period Arnold tongue is tangent to the Neimark-Sacker bifurcation set. Normally, the Arnold tongue is surrounded by tangent bifurcation and period-doubling bifurcation, but in this system, there are three bifurcations set as in the 3-period region of Fig. 2a.

Since the entire program is written in Python, the computation speed of this method is inferior to that of conventional programs written in C/C++ or Fortran. In fact, the calculation of the I^2 bifurcation point at $w_{11} = 2$ and $w12 = -1.92351$ in Fig. 2a can be done in about 1 ms using a C++ program with the Eigen library, while the proposed method requires about 800 ms. It is clear that the number of iterations required for convergence of the Newton method does not change. The CPU used for the calculation is AMD Ryzen5 1600. However, the shortcomings in the speed of the computation can be solved by using a GPU for matrix computation or by creating a mechanism to call the derivative preprocessor in Python from C++ or other programs.

4. Conclusion

In this paper, we propose a preprocessor for bifurcation analyses of discrete dynamical systems using symbolic differentiation. The proposed method is very useful for computing bifurcations of simple discrete dynamical systems, but its performance is observed to be poor for complex systems and multi-period points. One of the ways to deal with this problem is to write the derivatives of the iterative composite map into the program in advance, which enables bifurcation analysis while maintaining a convenient interface. As an application example, we performed the bifurcation analysis of a discrete



(a) $w_{21} = 5, w_{22} = -5, \xi = 1.5$.

(b) $w_{12} = -2, w_{21} = 5, \xi = 1.5$.

Fig. 2: Bifurcation diagram of Eq.(6).

neuron dynamical system using the Swish function as the activation function. Although the derivatives of this system become very complicated due to the activation function, the bifurcation structure can be investigated by using the proposed method. However, since the proposed method is implemented in Python, its performance is not as good as that of conventional bifurcation analysis packages written in C/C++ or Fortran. Our future work is to implement the variational equation preprocessor in other languages with high computational speed and applying the proposed method to high-dimensional systems.

References

[1] K. Tsumoto, T. Ueta, T. Yoshinaga, and H. Kawakami, "Bifurcation analyses of nonlinear dynamical systems: From theory to numerical computations," *Nonlinear Theory and Its Applications, IEICE*, vol. 3, no. 4, pp. 458–476, 2012.

[2] E. J. Doedal, "AUTO: Software for continuation and bifurcation problems in ordinary differential equations," `http://github.com/auto-07p`.

[3] Y. A. Kuznetsov, *Elements of applied bifurcation theory.* Applied Mathematical Sciences, Springer.

[4] T. Ueta, T. Yoshinaga, H. Kawakami, and G. Chen, "A method to calculate Neimark–Sacker bifurcation in autonomous systems," *IEICE Trans. Fundam.*, vol. 10, pp. 1141–1147, 2000.

[5] T. Kousaka, T. Ueta, and H. Kawakami, "Bifurcation of switched nonlinear dynamical systems," *IEEE Trans. Circuits and Systems II*, vol. 46, pp. 878–885, 1999.

[6] A. Solo, "Multidimensional matrix mathematics: Multidimensional matrix equality, addition, subtraction, and multiplication, part 2 of 6," *Proc. of the World Congress on Engineering 2010*, vol. 3, 06 2010.

[7] H. Nakajima, *Nonlinear Analyses of the Processes of Learning Dynamical Systems in Neural Network.* PhD thesis, Kyoto University, 1995.

[8] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," 2018, arXiv:1811.03378.

[9] X. Wang, "Period-doublings to chaos in a simple neural network," in *Proc. of IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. ii, pp. 333–339, 1991.

[10] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *CoRR*, vol. abs/1710.05941, 2017.