2023 International Symposium on Nonlinear Theory and Its Applications
NOLTA2023, September 26-29, 2023, Catania and Online

Python Expressions of Variational Equations

Tetsushi Ueta

Center for Administration of Information Technology
2-1 Minami-Josanjima, Tokushima 770-8506, Japan

Email: ueta@tokushima-u.ac.jp

Abstract— Python is gaining attention as a fundamental programming language for machine learning and data science. This paper describes a detailed Python approach to nonlinear problems, especially the bifurcation problems of periodic solutions. It is a highly readable implementation of the bifurcation algorithm, independent from the computer and the operating system, and it allows an interactive trial-and-error processing.

1. Introduction

Bifurcation phenomena are topological changes of equilibria or periodic solutions in the given dynamical system. The conditions of a bifurcation can be formulated by the simultaneous equations including a fixed point condition and the characteristic equation. To solve the equations accurately for the fixed point and the parameter value, Newton's method is reasonably applied since above all conditions are differentiable. Since these derivatives of the method are solutions of the variational equations about the fixed point, a numerical integration of both given differential equation and variational equations is required.

When a nonlinear dynamical system is given by differential equations or difference equations, a comprehensive analysis of the behavior of solutions in response to changes in parameters is necessary to understand its dynamic properties. As it is difficult to obtain analytical solutions for nonlinear systems in general, the solutions are computed numerically and, if a periodic solution is confirmed after the transient response, it will become the basis for analysis. Changes in parameters can lead to changes in the stability of these periodic solutions, namely bifurcation phenomena may be found. Manifolds that provide bifurcation phenomena in parameter space are called bifurcation sets, and the main subject of this tutorial is to find these as simply as possible. Maps on parameter space constructed by bifurcation sets (bifurcation diagrams) eloquently express the qualitative properties of the given dynamical system and will likely provide useful guidance to the objectives of individual problems. This tutorial shows a unique Python approach in the calculation of bifurcation sets. "Simply" in Python means using adverbs like short, condensed, and simple. It will increase readability and potentially prevent the introduction of bugs. An example of a non-autonomous system is taken to describe the Python implementation in the computation of bifurcation sets of periodic solutions.

Bifurcation phenomena refer to changes in the stability of equilibrium points and periodic solutions in a dynamical system, and there is a high demand to accurately determine the parameter values at which these occur. In general, it is difficult to gain prior knowledge about periodic solutions, let alone their bifurcation phenomena, based solely on the given equations. Some trial and error is necessary. We investigate which parameters of the differential equations have solutions of what period, and secondarily, with what initial values and through what transient responses we can reach these solutions.

Details about related algorithms and tools, including AUTO written by Doedel[2], are available in Kuznetsov's book[4]. This tutorial will primarily discuss the Python implementation of an algorithm developed by Kawakami, contemporaneous with the first version of AUTO. For essential differences between related algorithms, please refer to the relevant literature[4].

The tools developed in the lab based on specific algorithms would be invaluable. In order to maintain, improve, and expand them while reflecting new research trends, it's ideal to avoid as much as possible costs related to dependencies on computers or operating systems and securing and maintaining computational resources. Moreover, to obtain meaningful data through trial and error, it would be desirable to interactively operate the tool while it is running. Furthermore, apart from appearances on display, it is important to have visualization methods of a quality that can withstand use in academic papers.

Python is a cross-platform, open-source computing language that is gaining attention in the fields of data science and AI. The Python libraries, Numpy and Scipy, have code ported from the scientific computation libraries BLAS (Basic Linear Algebra Subprograms)[5] and LAPACK (Linear Algebra PACKage)[1], inherited from the FORTRAN era, and are highly reliable. Also, the graph drawing library, Matplotlib, allows for high-quality graphics to be used in interactive processing.

For Detailed explanation of computation algorithms for bifurcation sets, please refer Refs. [3] and [6]. In this paper, we focus on effectiveness, merits when the computation algorithm is written by Python. Especially, the worth of designing variational equations is highlighted.

ORCID iD TU: 0000-0001-5810-437X

1.1. Preliminaries

Consider an initial value problem:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(t, \boldsymbol{x}, \lambda), \quad \text{with} \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \tag{1}$$

where, $\boldsymbol{x} \in \boldsymbol{R}^n$ is the state, $\lambda \in \boldsymbol{R}$ is a parameter. We assume that $\boldsymbol{f} : \boldsymbol{R}^n \to \boldsymbol{R}^n$ is C^∞-class and a τ-periodic function such that $\boldsymbol{f}(t+\tau, \boldsymbol{x}, \lambda) = \boldsymbol{f}(t, \boldsymbol{x}, \lambda)$. We define a solution starting from $\boldsymbol{x}_0$ at $t = 0$ as $\boldsymbol{x}(t) = \boldsymbol{\varphi}(t, \boldsymbol{x}_0, \lambda)$ satisfying $\boldsymbol{x}(0) = \boldsymbol{\varphi}(0, \boldsymbol{x}_0, \lambda) = \boldsymbol{x}_0$. If there is a periodic solution in Eq. (1), it is expressed as $\boldsymbol{\varphi}(0, \boldsymbol{x}_0, \lambda) = \boldsymbol{\varphi}(\tau, \boldsymbol{x}_0, \lambda)$.

For this periodic solution, we apply the Poincaré map such as:

$$\begin{aligned} T: \quad & \boldsymbol{R}^n \to \boldsymbol{R}^n \\ & \boldsymbol{x}_0 \mapsto T(\boldsymbol{x}_0) = \boldsymbol{\varphi}(\tau, \boldsymbol{x}_0, \lambda). \end{aligned} \tag{2}$$

This samples a point every t from the orbit like $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \ldots\}$, thus it naturally gives a discrete difference equation such as:

$$\boldsymbol{x}_{k+1} = T(\boldsymbol{x}_k). \tag{3}$$

The periodic solution of Eq. (1) is corresponding to the fixed point $\boldsymbol{x}_0$ such as $\boldsymbol{x}_0 = T(\boldsymbol{x}_0)$. For $\ell \geq 2$, if $\boldsymbol{x}_1 = T(\boldsymbol{x}_0)$, $\boldsymbol{x}_2 = T(\boldsymbol{x}_1), \ldots, \boldsymbol{x}_0 = T(\boldsymbol{x}_{\ell-1})$, i.e., $\boldsymbol{x}_0 = T^\ell(\boldsymbol{x}_0)$ is satisfied, $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{\ell-1}\}$. gives periodic points.

The variational equations are subsequent equation for the given differential equation, and are derivatives of the solutions by applying the chain-rule[3].

2. Computation of the fixed point

The condition of the fixed point is formulated by a two-point boundary problem:

$$T(\boldsymbol{x}_0) - \boldsymbol{x}_0 = \boldsymbol{0} \tag{4}$$

For the ℓ-periodic point, this condition is also available by identifying T^ℓ as T.

To compute the fixed point $\boldsymbol{x}_0$ accurately, Newton's method is the priority choice since it converges quadratically if Eq. (4) is differentiable. When implementation, the Jacobian matrix of Eq. (4) for $\boldsymbol{x}_0$ is required. It is given by concretely:

$$\frac{\partial}{\partial \boldsymbol{x}_0}\Big(T(\boldsymbol{x}_0) - \boldsymbol{x}_0\Big) = \left.\frac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0}\right|_{t=\tau} - \boldsymbol{I}_n, \tag{5}$$

where $\boldsymbol{I}_n$ is an $n \times n$ identity matrix. According to this Jacobian matrix, the characteristic equation is given by the following equation:

$$\chi(\mu) = \det\left(\left.\frac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0}\right|_{t=\tau} - \mu \boldsymbol{I}_n \right) = 0. \tag{6}$$

The characteristic multiplier, which is the n roots of this equation, indicates the stability of the fixed point. $\partial\boldsymbol{\varphi}/\partial\boldsymbol{x}_0$ is called a variation appeared both Eqs. (5) and (6). The problem is how to find this variation.

Since the variations are the derivatives of the solution, when the solution trajectories of the equation (1) are obtained numerically, it is possible to substitute their numerical derivatives as the variations. For example, if $\Delta > 0$ is sufficiently small and $\boldsymbol{x}_0 = (x_{01}, x_{02}, \ldots, x_{0n})^\top$, then We can approximate the variation vector with respect to x_{01} by the formula:

$$\begin{aligned} &\frac{\partial \boldsymbol{\varphi}}{\partial x_{01}}(\tau, x_{01}, x_{02}, \ldots, x_{0n}) \approx \\ &\quad \frac{1}{\Delta}\Big(\boldsymbol{\varphi}(\tau, x_{01} + \Delta, x_{02}, \ldots, x_{0n}) \\ &\qquad -\boldsymbol{\varphi}(\tau, x_{01}, x_{02}, \ldots, x_{0n})\Big) \end{aligned}$$

However, since numerical differentiation could essentially contain errors, it may affect the value of the characteristic multiplier and the convergence performance of Newton's method, so we want to avoid using it as much as possible.

2.1. Variations by initial values

Now, by substituting the solution into the original differential equation (1) and changing the order of differentiation, we obtain the first variational equation, which is a variable-coefficient linear differential equation[3]:

$$\frac{d}{dt}\frac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0}, \quad \text{with} \quad \left.\frac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0}\right|_{t=0} = \boldsymbol{I}_n, \tag{7}$$

where, $\partial\boldsymbol{f}/\partial\boldsymbol{x}$ is the Jacobian matrix of Eq. (1) (differentiating the right hand of Eq.(1) symbolically and partially with the state variable $\boldsymbol{x}$). More precisely, Eq. (7) is rewritten as follows.

$$\begin{aligned} &\frac{d}{dt}\left(\begin{array}{c|c|c|c} \frac{\partial \varphi_1}{\partial x_{01}} & \frac{\partial \varphi_1}{\partial x_{02}} & \cdots & \frac{\partial \varphi_1}{\partial x_{0n}} \\ \frac{\partial \varphi_2}{\partial x_{01}} & \frac{\partial \varphi_2}{\partial x_{02}} & \cdots & \frac{\partial \varphi_2}{\partial x_{0n}} \\ \vdots & \vdots & & \vdots \\ \frac{\partial \varphi_n}{\partial x_{01}} & \frac{\partial \varphi_n}{\partial x_{02}} & \cdots & \frac{\partial \varphi_n}{\partial x_{0n}} \end{array}\right) = \\ &\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \left(\begin{array}{c|c|c|c} \frac{\partial \varphi_1}{\partial x_{01}} & \frac{\partial \varphi_1}{\partial x_{02}} & \cdots & \frac{\partial \varphi_1}{\partial x_{0n}} \\ \frac{\partial \varphi_2}{\partial x_{01}} & \frac{\partial \varphi_2}{\partial x_{02}} & \cdots & \frac{\partial \varphi_2}{\partial x_{0n}} \\ \vdots & \vdots & & \vdots \\ \frac{\partial \varphi_n}{\partial x_{01}} & \frac{\partial \varphi_n}{\partial x_{02}} & \cdots & \frac{\partial \varphi_n}{\partial x_{0n}} \end{array}\right) \end{aligned} \tag{8}$$

Note that the both sides of this equation can be divided into the first order simultaneous differential equation. Therefore, one can obtain the fundamental matrix $\partial\boldsymbol{\varphi}/\partial\boldsymbol{x}_0|_{t=\tau}$ by integrating the initial value problem (7) from $t = 0$ to $t = \tau$ numerically with an appropriate solver like `solve_ivp()`. This can be utilized in Eqs (5) or (6) in Eq. (7). For the right hand of the differential equation (1), the result of applying the differentiation and the chain rule by the initial value is arranged as a product of matrices.

Let us consider the implementation of Eq. (7) with Python. Provide a two-dimensional `ndarray` for the Jacobian matrix `dfdx`, and variations `dphidx`, a Python code for the right hand of Eq. (8) is given by:

```
dfdx @ dphidx
```

where, '@' is a new product operator which has been released at the version 3.5 and it is a macro for matmul function. Since `solve_ivp()` requires vectors for input/output, therefore, we divide the matrix as the result of computation for the right hand of Eq. (8) into n column vectors, and flatten into a single vector. Then append this vector into the `func` (the right hand vector of Eq. (1). This procedure is expressed as:

```
func.extend((dfdx @ dphidx).T.flatten())
```

where, `.T` attribute is a transpose, and `.flatten()` is a flattening method. For other programming languages, we should need several nest of loops for these operations and multiplications, however, Python can express them as one line.

3. Computation of bifurcation parameter sets

For computation of bifurcation parameter values, we simultaneously solve Eqs. (4), (6) for $\boldsymbol{x}_0$ and λ, that is,

$$\begin{cases} T(\boldsymbol{x}_0) - \boldsymbol{x}_0 = \boldsymbol{0} \\ \chi(\mu) = 0 \end{cases} \tag{9}$$

where, we put a specific value to μ corresponding bifurcation phenomenon, e.g., for the tangent bifurcation, we put $\mu = 1$.

If the shape of the solution trajectory changes significantly when the parameters are changed in the phase portrait, the fixed points and characteristic multipliers are calculated by using the fixed point calculation tool described in the previous section. When the characteristic multiplier is close to the above specific value, taking that $(\boldsymbol{x}_0, \lambda)$ as an initial guess, we could get bifurcation parameter value by solving Eq. (9) with Newton's method.

The Jacobian matrix of Newton's method is the following formulation.

$$\left(\begin{array}{c|c} \dfrac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0} - \boldsymbol{I}_n & \dfrac{\partial \boldsymbol{\varphi}}{\partial \lambda} \\ \hline \dfrac{\partial \chi}{\partial \boldsymbol{x}_0} & \dfrac{\partial \chi}{\partial \lambda} \end{array}\right) \tag{10}$$

Compared with the fixed-point computation, this scheme requires the second variation regarding the initial value and the parameter. However, we want to avoid using the numerical differentiation since it may include errors.

3.1. Variations by parameters

The variations by the parameter value about the fixed point is given by the numerical integration (solving ODEs) for the following n-dimensional variable coefficient linear non homogeneous differential equations:

$$\frac{d}{dt}\frac{\partial \boldsymbol{\varphi}}{\partial \lambda} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{\varphi}}{\partial \lambda} + \frac{\partial \boldsymbol{f}}{\partial \lambda}, \quad \text{with} \quad \left.\frac{\partial \boldsymbol{\varphi}}{\partial \lambda}\right|_{t=0} = \boldsymbol{0} \tag{11}$$

Let us describe this formula by Python. Define the variations by the parameter as `dphidl` and assume a vector `dfdl` which is obtained by the partial differentiation for Eq. (1) by a parameter symbolically, the computation of Eq. (11) and appending the result into the vector `func` for `solve_ivp()` is described as:

```
func.extend(dfdx @ dphidl + dfdl)
```

In the parentheses, for a product of a matrix and a vector, the operator '@' is also available.

3.2. Second variations by initial values

By partially differentiate Eq. (7) by $\boldsymbol{x}_0$, and arrange the chain-rule, we have n^3-tuple ODEs. We call this the second variational equations. Let $\partial^2\boldsymbol{\varphi}/\partial\boldsymbol{x}_0^2$ be a n-dimensional third-order tensor, the second variational equations are written as:

$$\begin{aligned} \frac{d}{dt}\frac{\partial^2 \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0^2} &= \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\frac{\partial^2 \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0^2} + \frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}^2}\left(\frac{\partial \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0}\right)^2, \\ &\text{with} \quad \left.\frac{\partial^2 \boldsymbol{\varphi}}{\partial \boldsymbol{x}_0^2}\right|_{t=0} = \boldsymbol{O}, \end{aligned} \tag{12}$$

where, $\boldsymbol{O}$ is an $n \times n \times n$ tensor whose all elements are zero. $\partial^2\boldsymbol{f}/\partial\boldsymbol{x}^2$ is a Hessian tensor which is given by symbolically partial differentiation of the Jacobian matrix $\partial\boldsymbol{f}/\partial\boldsymbol{x}$ by the state variable $\boldsymbol{x}$. While, $\partial\boldsymbol{\varphi}/\partial\boldsymbol{x}_0$ is obtained by the numerical integration of Eq. (7) The right hand of the second term is described by this expression at the convenience sake, however, the computation of a product for a tensor and a matrix gives a complex procedure. In fact, if we consider the case of $n = 2$, a some part of the product should be written as:

$$\frac{d}{dt}\begin{pmatrix} \dfrac{\partial^2 \varphi_1}{\partial x_0^2} \\ \dfrac{\partial^2 \varphi_2}{\partial x_0^2} \end{pmatrix} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\begin{pmatrix} \dfrac{\partial^2 \varphi_1}{\partial x_0^2} \\ \dfrac{\partial^2 \varphi_2}{\partial x_0^2} \end{pmatrix} + \frac{\partial}{\partial x_0}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\begin{pmatrix} \dfrac{\partial \varphi_1}{\partial x_0} \\ \dfrac{\partial \varphi_2}{\partial x_0} \end{pmatrix} \tag{13}$$

where,

$$\begin{pmatrix} \dfrac{\partial^2 f_1}{\partial x^2}\dfrac{\partial \varphi_1}{\partial x_0} + \dfrac{\partial^2 f_1}{\partial y \partial x}\dfrac{\partial \varphi_2}{\partial x_0} & \dfrac{\partial^2 f_1}{\partial x \partial y}\dfrac{\partial \varphi_1}{\partial x_0} + \dfrac{\partial^2 f_1}{\partial y^2}\dfrac{\partial \varphi_2}{\partial x_0} \\ \dfrac{\partial^2 f_2}{\partial x^2}\dfrac{\partial \varphi_1}{\partial x_0} + \dfrac{\partial^2 f_2}{\partial y \partial x}\dfrac{\partial \varphi_2}{\partial x_0} & \dfrac{\partial^2 f_2}{\partial x \partial y}\dfrac{\partial \varphi_1}{\partial x_0} + \dfrac{\partial^2 f_2}{\partial y^2}\dfrac{\partial \varphi_2}{\partial x_0} \end{pmatrix}. \tag{14}$$

This seems to be difficult to check its validity. Indeed, for Eq. (12), letting $\boldsymbol{f} = (f_1, f_2, \ldots, f_n)^\top$ and $\boldsymbol{\varphi} = (\varphi_1, \varphi_2, \ldots, \varphi_n)^\top$, the extracted expression of the general term in the second variational equation is as follows:

$$\begin{aligned}\frac{d}{dt}\frac{\partial^2\varphi_i}{\partial x_{0k}\partial x_{0\ell}} &= \sum_{p=1}^{n}\frac{\partial f_i}{\partial x_p}\frac{\partial^2\varphi_p}{\partial x_{0k}\partial x_{0\ell}} \\ &+\sum_{p=1}^{n}\sum_{q=1}^{n}\frac{\partial^2 f_i}{\partial x_p\partial x_q}\frac{\partial\varphi_p}{\partial x_{0k}}\frac{\partial\varphi_q}{\partial x_{0\ell}}, \\ \text{with}\quad & \left.\frac{\partial^2\varphi_i}{\partial x_{0k}\partial x_{0\ell}}\right|_{t=0} = 0 \\ \text{for}\quad & i,k,\ell = 1,2,\ldots,n\end{aligned} \tag{15}$$

The total number of these equations becomes n^3. This expression is known as a formula, but we are not sure that one can rely on this without any verification. The algorithm requires multiple-nest loops to interpret this term, and we have to check carefully the indices and loop counters to avoid the mistakes.

Let us describe the right hand of Eq. (12) by Python.

```
dfdx@d2phidx2 + (d2fdx2@dphidx).T @ dphidx
```

where, d2phidx2 and d2fdx2 are the second variations and a Hessian, respectively. No loops, slices, indices, loop counters are required. An intuitive expression for the right hand of Eq. (12) gives a proper computation of the second variations. Provide P as this result, the extraction and concatenation as a vector is given by:

```
P.transpose(0,2,1).flatten()
```

The point is changing the order of the axes for a transpose operation.

From the Yang's theorem, exchanging the order for 2-variable partial differentiation only requires a half numbers of n^3 tuple ODEs since the symmetric property: $\partial^2\boldsymbol{\varphi}_i/\partial x_k\partial x_\ell = \partial^2\boldsymbol{\varphi}_i/\partial x_\ell\partial x_k$. Thus we have:

List 1: Computation for the right hand of Eq. (12)

```
1   ui, uj = np.triu_indices(n)
2   v = x.reshape(int(n*(n+1)/2), n)
3   X = np.zeros(n**3).reshape(n, n, n)
4   X[ui, uj] = v
5   X[uj, ui] = v
6   d2phidx2 = X.transpose(0, 2, 1)
7   P = (dfdx @ d2phidx2 + (d2fdx2 @ dphidx).T
        @ dphidx).transpose(0, 2, 1)
8   func.extend(P[ui, uj].flatten())
```

List 1 shows the flow of passing only the minimum necessary second variation to solve_ivp(). The first line gives the row and column index lists of valid elements in the upper triangular matrix. When the minimum necessary number of second variations is already stored in the vector x, we convert it back into a symmetric second variation tensor d2phidx2 (rows 2–7) by specifying the index list of the upper triangular matrix. No loops, no slice operations required. After that, we complete the computation of the second variations (line 7) and pass only the optimized number of the second variations to solve_ivp() (line 8).

3.3. Second variations by parameters

The last item we have to prepare for Newton's method is the second variations for a parameter. By doing a partial differentiation for Eq. (7) by λ and applying the chain rule, we have:

$$\begin{aligned}\frac{d}{dt}\frac{\partial^2\boldsymbol{\varphi}}{\partial\boldsymbol{x}_0\partial\lambda} &= \frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}}\frac{\partial^2\boldsymbol{\varphi}}{\partial\boldsymbol{x}_0\partial\lambda} + \frac{\partial^2\boldsymbol{f}}{\partial\boldsymbol{x}^2}\frac{\partial\boldsymbol{\varphi}}{\partial\boldsymbol{x}}\frac{\partial\boldsymbol{\varphi}}{\partial\lambda} + \frac{\partial^2\boldsymbol{f}}{\partial\boldsymbol{x}\partial\lambda}\frac{\partial\boldsymbol{\varphi}}{\partial\boldsymbol{x}_0} \\ \text{with}\quad & \left.\frac{\partial^2\boldsymbol{\varphi}}{\partial\boldsymbol{x}_0\partial\lambda}\right|_{t=0} = \boldsymbol{O}\end{aligned} \tag{16}$$

where, $\boldsymbol{O}$ is an $n \times n$ zero matrix. The right hand of Eq. (16) is expressed as follows by Python:

```
func.extend(dfdx @ d2phidxdl
  + ((d2fdx2 @ dphidx).T @ dphidl).T
  + (d2fdxdl @ dphidx)).T.flatten()
```

where, dphidl should be set by solving Eq. (11) in advance.

Now we are ready to solve Eq. (9) by Newton's method. Accurate location of the fixed point and the bifurcation parameter value are obtained within a couple of iterations. Bifurcation sets for the unstable periodic point which is unable to visualize by the brute-force method is able to computed. For more details about computations of bifurcation sets, please refer Ref.[7].

4. Conclusions

We try to express variational equations by Python. Very simple codes can exclude a possibility of happening bugs, insertion of vulnerable codes. For the product operation for tensors, even einsum() operator can describe any rule of productions, but we particularly confirmed that the default operation of production operatior in Python is reasonable for implementation of the chain-rule for variational equations.

Acknowledgments

This work is supported partially by KAKENHI grant number 21K04109 and JST Moonshot R& D grant number JPMJMS2021.

References

[1] E. Angerson et al. LAPACK: A portable linear algebra library for high-performance computers. In *Proc. ACM/IEEE Conf. Supercomputing*, pages 2–11, 1990.

[2] E. J. Doedel. AUTO: A program for the automatic bifurcation analysis of autonomous systems. *Congr. Numer*, 30:265–284, 1981.

[3] H. Kawakami. Bifurcation of periodic responses in forced dynamic nonlinear circuits: Computation of bifurcation values of the system parameters. *IEEE Trans. Circuits Syst.*, 31(3):248–260, 1984.

[4] Y. A. Kuznetsov. *Elements of applied bifurcation theory*, volume 112. Springer, third edition, 2004.

[5] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979.

[6] K. Tsumoto, T. Ueta, T. Yoshinaga, and H. Kawakami. Bifurcation analyses of nonlinear dynamical systems: From theory to numerical computations. *NOLTA*, 3(4):458–476, 2012.

[7] T. Ueta and S. Amoh. To tacke bifurcation problems with Python (in Japanese). *IEICE ESS Fundamentals Review*, 16(3):139–146, 2023.