Kaa: A Python Implementation of Reachable Set Computation Using Bernstein Polynomials

Edward Kim and Parasara Sridhar Duggirala

Department of Computer Science
University of North Carolina at Chapel Hill
{ehkim,psd}@cs.unc.edu

Abstract

Reachable set computation is one of the widely used techniques for verification of safety properties of dynamical system. One of the simplest algorithms for computing reachable sets for discrete nonlinear systems uses parallelotope bundles and Bernstein polynomials. In this paper, we describe Kaa, a terse Python implementation of reachable set computation which leverages the widely used symbolic package sympy. Additionally, we simplify the user interface with the tool and provide easy to use plotting utilities. We believe that our tool has pedagogical use given the simplicity of the implementation and its user-friendliness.

Nonlinear Dynamical Systems; Reachable Set Computation; Bernstein Polynomials.

1 Introduction

Reachable set computation is one of the many important tools available for verification of dynamical and hybrid systems. One of the simpler and easier-to-understand reachable set computation algorithms utilizes Bernstein polynomials and parallelotopes [2]. Several cumulative improvements [3, 11, 1, 5, 6] have been made to improve the accuracy and efficiency of nonlinear systems reachability. The tool SAPO [4] which implements these algorithms in C++ is also available publically. In this paper, we present a Python implementation of algorithm presented in [6].

Our primary motivation for reimplementing the algorithms in Python is two-fold. First, using some of the standard libraries for symbolic manipulation in Sympy, the algorithm for reachability can be implemented very concisely. In fact, Kaa is roughly only 650 lines of code. Additionally, the main algorithm for computing the reachable set is provided in just one file with 150 lines of code. In contrast, SAPO is more than 2000 lines of C++ code with memory and pointer management performed completely by the developer. We believe that such a simple implementation has pedagogical value and can be used for teaching one of the main algorithms for reachability of nonlinear systems. Second, reimplementing the tool in Python makes user interaction significantly easy. In SAPO, for computing the reachable set, one has to recompile the code along with the model file to obtain the binary and execute the binary to compute reachable set. Additionally, SAPO generates a separate MATLAB script for visualizing time or phase plots of the reachable set. This script must be fed separately into either MATLAB or

Octave to illustrate the reachable set. In contrast, we provide a Jupyter notebook interface for Kaa to create an intuitive platform facilitating experimentation. Juypter notebooks are simple to create and known for their straightforward user interface. By leveraging the matplotlib library for visualizing the reachable set, we were able to design an engaging interactive tutorial and experimentation platform for visualzing reachable sets of non-linear systems.

Bernstein polynomials are also considered an active area of research in the domain of global optimization [10, 7, 9]. There have been several algorithms proposed for performing heuristics which improve the performance of optimization using Bernstein polynomials. Furthermore, these algorithms also improve the accuracy using sequential refinement techniques [12, 8]. Having an accessible and flexible tool would make it more conducive to implement these heuristics and determine specific heuristics helpful in the domain of reachability. In light of these features and advantages, we believe Kaa can be used as a easy first step for introducing reachability due to its simplicity of implementation and its usability.

2 Preliminaries

The state of a system, denoted as x, lies in a domain $D \subseteq \mathbb{R}^n$. A discrete-time polynomial nonlinear system is denoted as

$$x^+ = f(x) \tag{1}$$

where $f(x) : \mathbb{R}^n \to \mathbb{R}^n$ is polynomial in x. The trajectory of the system that evolves according to Equation (1), denoted as $\xi(x_0)$, is the sequence $x_0, x_1, \ldots$ where $x_{i+1} = f(x_i)$. The k^{th} element in this sequence is denoted as $x_k = \xi(x_0, k)$. Given an initial set Θ, the reachable set at time k, denoted as $\Theta_k = \{\ \xi(x_0, k) \mid x_0 \in \Theta\}$.

A parallelotope P is a set of states in $\mathbb{R}^n$ denoted as $\langle \Lambda, c\rangle$ where $\Lambda \in \mathbb{R}^{2n\times n}$ and $c \in \mathbb{R}^{2n}$, $\Lambda_{i+n} = -\Lambda_i$ and $i \in \{1, \ldots, n\}$ such that

$$x \in P \text{ if and only if } \Lambda x \leq c.$$

Λ is called the *direction matrix* Λ_i denotes the i^{th} row of Λ. c is called the *offset*. For a parallelotope P, we denote the affine transformation that maps $[0,1]^n$ to P as T_p. A parallelotope bundle Q is a set of parallelotopes $\{P_1, \ldots, P_m\}$ where $Q = \cap_{i=1}^{m} P_i$.

Given two multi-indices i and d of size n, where $i \leq d$, the Bernstein polynomial of degree d and index i is

$$\mathcal{B}_{i,d} = \beta_{i_1,d_1}(x_1)\beta_{i_2,d_2}(x_2)\ldots\beta_{i_n,d_n}(x_n).$$

where $\beta_{i_m,d_m}(x_m) = \binom{d_m}{i_m} x_m^{i_m}(1-x_m)^{d_m-i_m}$. Any polynomial function can be expressed in the Bernstein basis. The primary advantage of the Bernstein representation of a polynomial $p(x_1, ..., x_n)$ is that an upper bound on the supremum and lower bound on the infimum of $p(x_1, ..., x_n)$ in $[0,1]^n$ can be computed purely by observing the coefficients of the polynomial in the Bernstein basis.

In other words, given a polynomial $p(x_1, \ldots, x_n) = \sum_{j\in J} a_j \mathbf{x}_j$ where J is a set of multi-indices iterating through the degrees found in p with $a_j \in \mathbb{R}$, then $p(x_1, \ldots, x_n)$ can be converted into its counterpart under the Bernstein basis, $p(x_1, \ldots, x_n) = \sum_{j\in J} b_j \mathcal{B}_j$ where b_j are the corresponding Bernstein coefficients. The upper and lower bounds of $p(x_1, \ldots, x_n)$ over $[0,1]^n$ are exactly bounded by the Bernstein coefficients:

$$min\{b_1, \ldots, b_m\} \ \leq \ inf_{x\in[0,1]^n} h(x) \quad \leq \quad sup_{x\in[0,1]^n} h(x) \ \leq \ max\{b, \ldots, b_m\}.$$

Any parallelotope P can be considered as a suitable affine transformation T_p from $[0,1]^n$ to P. Therefore, upper bounds on the suprenum of a function h over P is equivalent to upper bound of $h \circ T_p$ over $[0,1]^n$. A similar argument follows for the lower bound on infimum. For the remainder of the document, we assume that by using functional composition and the Bernstein representation, we can easily compute the upper bound on supremum and lower bound on infimum of polynomial functions over parallelotopes.

Furthermore, we denote the procedures for calculating such upper and lower bounds for a polynomial p over some parallelotope S as $\mathsf{BernsteinUpper}(p, S)$ and $\mathsf{BernsteinLower}(p, S)$ respectively.

3 Reachability of Nonlinear Systems Using Parallelotope Bundles and Bernstein Polynomials

Given a set represented as a parallelotope bundle $Q = \{P_1, P_2, \ldots, P_m\}$ and a discrete dynamical system $x^+ = f(x)$, we now present the method for computing an overapproximation of $f(Q)$ as a parallelotope bundle $Q' = \{P'_1, P'_2, \ldots, P'_m\}$. We exert that the direction matrix of P'_i is same as P_i and the computation is required only to compute the offsets. Denote the j^{th} offset of P_i and P'_i as $c_{j,i}$ and $c'_{j,i}$ respectively and the j^{th} direction in P_i (same as P'_i) as $\Lambda_{j,i}$. Therefore, given Λ, transformation f, and offsets $c_{j,i}$ for P_i, we have to compute the values of $c'_{j,i}$.

If $j \leq n$, any offset $c'_{j,i}$ such that $\forall_{x \in Q} \Lambda_{j,i} \cdot f(x) \leq c'_{j,i}$ is valid. If $j > n$, then any offset $c'_{j,i}$ such that $\forall_{x \in Q} \Lambda_{j-n,i} \cdot f(x) \geq c'_{j,i}$ is valid. Therefore, one can compute the j^{th} offset of P'_i, i.e., $c'_{j,i}$ for $j \leq n$ by computing an upper bound of the function $\Lambda_{j,i} \cdot f(x)$ over the $x \in P_i$. Similarly, the $j + n^{th}$ offset can be computed from a lower bound of the function $\Lambda_{j,i} \cdot f(x)$ over each parallelotope $x \in P_i$. This is given in Equations 2 and 3.

$$c_{j,i} = min_{l=1}^{m} \Big\{ \mathsf{BernsteinUpper}\big(\Lambda_{j,i} \cdot f(x), P_l\big) \Big\} \text{ if } j \leq n. \quad (2)$$

$$c_{j+n,i} = max_{l=1}^{m} \Big\{ \mathsf{BernsteinLower}\big(\Lambda_{j,i} \cdot f(x), P_l\big) \Big\} \text{ otherwise.} \quad (3)$$

3.1 Python Implementation of Reachable Set Computation

One of the primary reasons for preferring Python as an implementation language for this algorithm is the presence of powerful, well-tested symbolic and matrix-computation libraries. Python's *numpy* libraries are popular matrix-computation libraries which allow higher-level manipulation of matrices. This gives us an avenue of overcoming the verbosity and the possibility of memory leaks inherent in implementing identical features in C++. We believe that overcoming these obstacles results in a more readable, more compact approach to parallelotope reachability suitable for graduate students and curious practitioners. In addition, these libraries are known for their extensive set of useful functionalities which allow us to avoid reimplementing many fundamental operations.

As we have seen before, there are two main sub-routines for computing the reachable set using Bernstein polynomials. First, performing functional composition for computing uthe pper bound of a polynomial over a parallelotope. Second, computing the Bernstein representation of a polynomial. The library of *sympy* has powerful symbolic manipulation tools which allow us to comfortably perform many sensitive symbolic subsititutions into polynomials. We use *sympy*'s internal representation of polynomials to (a) perform functional composition and (b) compute the Bernstein representations of the resulting polynomial. Additionally, the *matplotlib* library has easy plotting facilities that we integrate into our tool for visualizing the reachable set. In particular, *matplotlib* facilitates the ability to visualize several reachable sets simultaneously. This will aid others in performing more complex bundle-transformation experiments in future.

Finally, Python has a rich set of multiprocessing libraries, namely *multiprocessing* or *mp* for short. In the future, we plan to exploit the parallelizable nature of bundle-transformtion computations using *mp*'s ability to provide powerful multiprocessing features without much verbosity.

4 Evaluations

We evaluate our tool on the benchmarks that are shipped along with SAPO. With each benchmark, the reachable sets of SAPO are juxtaposed with that of Kaa's to demonstrate the accuracy and quality of our reachable set. A comparison of time taken by Kaa and SAPO for each benchmark is provided in Table 3.

4.1 SIR Epidemic Model

The SIR Epidemic model is a 3-dimensional dynamical system governed by the following dynamics:

$$\begin{aligned} s_{k+1} &= s_k - (\beta s_k i_k)\Delta \\ i_{k+1} &= i_k + (\beta s_k i_k - \gamma i_k)\Delta \\ r_{k+1} &= r_k + (\gamma i_k)\Delta \end{aligned} \tag{4}$$

where s, i, r represent the fractions of a population of individuals designated as *susceptible*, *infected*, and *recovered* respectively. There are two parameters, namely β and γ, which influence the evolution of the system. β is labeled as the contraction rate and $1/\gamma$ is mean infective period. Finally, Δ is simply the discretization step. For the benchmarks, we set $\beta = 0.34$, $\gamma = 0.05$, and $\Delta = 0.5$. The reachable sets are displayed in Figure 1. The table of numerical value comparisions between Sapo and Kaa is shown in Table 1.

Time Steps	**Kaa (offu)**	**Sapo (offu)**	**Kaa (offl)**	**Kaa (offl)**
50	0.470716	0.470716	-0.435191	-0.43519
51	0.475839	0.475839	-0.439599	-0.439599
52	0.480906	0.480906	-0.443945	-0.443945
53	0.485915	0.485915	-0.448227	-0.448227
54	0.490862	0.490862	-0.452443	-0.452443
55	0.495747	0.495747	-0.456591	-0.456591
56	0.500566	0.500566	-0.460669	-0.460669
57	0.505317	0.505317	-0.464675	-0.464675
58	0.509999	0.509999	-0.4686075	-0.468608
59	0.514610	0.514610	-0.472465	-0.472465
60	0.519147	0.519147	-0.476246	-0.476246

Table 1: Comparision of offu, offl values along variable i for the SIR model. We select the steps 50-60

Here, offu is the vector of upper offsets of the parallelotope and offl is the vector for the lower offsets. We define the upper facets of parallelotope P as the c_i for $i \leq n$ where n is the dimension of the system and $P = \langle \Lambda, c \rangle$ is the parallelotope. Similarly, the lower facets are defined as the c_{i+n} for $i \leq n$. In the case above, we are looking at values in c_2 and $c_{2+3} = c_5$.

4.2 Rossler Model

The Rossler model is another 3-dimensional system governed under the dynamics:

$$\begin{aligned} x_{k+1} &= x_k + -(y - z)\Delta \\ y_{k+1} &= y_k + (x_k + ay_k)\Delta \\ z_{k+1} &= z_k + (b + z_k(x_k - c))\Delta \end{aligned} \tag{5}$$

where a, b, c are parameters which we set to $a = 0,1$, $b = 0.1$, and $c = 14$. We set our discretization step to be $\Delta = 0.025$. The reachable sets are displayed in Figure **??** and the table showing the comparisions between the numerical values are shown in Table 2.

Time Steps	Kaa (offu)	Sapo (offu)	Kaa (offl)	Kaa (offl)
50	1.95908	1.96043	-1.9209	-1.9193
51	1.83552	1.83688	-1.7963	-1.7947
52	1.71044	1.71181	-1.67016	-1.6685
53	1.58392	1.58531	-1.54255	-1.5409
54	1.45604	1.45744	-1.41355	-1.4119
55	1.32687	1.32829	-1.28324	-1.2815
56	1.19649	1.19792	-1.15168	-1.1500
57	1.06499	1.06642	-1.01896	-1.0172
58	0.932432	0.933877	-0.885157	-0.88339
59	0.798905	0.800358	-0.75035	-0.74857
60	0.664489	0.665949	-0.614619	-0.61282

Table 2: Comparision of offu, offl values along variable y for the Rossler model. We select the steps 50-60

4.3 Quadcopter Model

The Quadcopter model is a 17-dimensional dynamical system with the following variables: A set of inertial positions: (p_n, p_e, h), linear velocities (u, v, w), the quaternions expressing the Euler angles (q_0, q_1, q_2, q_3), the angular velocities (p, q, r), and finally the parameters (h_I, u_I, v_I, ψ_I). The complete set of dynamics and relevant parameters is found in [6] and reachable sets are shown in Figure 5.

4.4 Lotka-Volterra Model

We also test on the competetitive Lotka-Volterra models for predator-prey biological systems. Our instance is a 5-dimensional sysmtem governed as below:

$$\begin{aligned} x_{k+1} &= x_k + (x_k(1-(x_k+\alpha y_k+\beta \ell_k)))\Delta \\ y_{k+1} &= y_k + (y_k(1-(y_k+\alpha z_k+\beta x_k)))\Delta \\ z_{k+1} &= z_k + (z_k(1-(z_k+\alpha h_k+\beta y_k)))\Delta \\ h_{k+1} &= h_k + (h_k(1-(h_k+\alpha \ell_k+\beta z_k)))\Delta \\ \ell_{k+1} &= \ell_k + (\ell_k(1-(\ell_k+\alpha x_k+\beta h_k)))\Delta \end{aligned} \tag{6}$$

where α, β are parameters set to $\alpha = 0.85$ and $\beta = 0.5$. We set $\Delta = 0.01$. The relevant reachable sets are shown in Figure **??**.

4.5 Phosphoraley Model

The Phosphoraley model describes a certain cellular regulatory system. It is captured by seven variables governed by the following dynamics:

$$\begin{aligned} x^1_{k+1} &= x^1_k + (-\alpha x^1_k + \beta x^3_k x^4_k)\Delta \\ x^2_{k+1} &= x^2_k + (\alpha x^1_k - x^2_k)\Delta \\ x^3_{k+1} &= x^3_k + (x^2_k \beta x^3_k x^4_k)\Delta \\ x^4_{k+1} &= x^4_k + (\beta x^5_k x^6_k - \beta x^3_k x^4_k)\Delta \\ x^5_{k+1} &= x^5_k + (-\beta x^5_k x^6_k + \beta x^3_k x^4_k)\Delta \\ x^6_{k+1} &= x^6_k + (\alpha x^7_k - \beta x^5_k x^6_k)\Delta \\ x^7_{k+1} &= x^7_k + (-\alpha x^7_k + \beta x^5_k x^6_k)\Delta \end{aligned} \tag{7}$$

where α, β are two parameters assigned as $\alpha = 0.5$ and $\beta = 5$. We set $\Delta = 0.01$ here. The relevant figures can be found in Figure **??**.

4.6 Times

It is evident from Table 3 that while implementation in Python is very intuitive and concise, the current implementation incurs severe performance penalties. Nevertheless, we pursued this activity because of the pedagogical value in making this an accessible tool for implementing and understanding reachability. To this end, the repository contains a Jupyter notebook with the name kaa-intro designed to interactively introduce graduate students and practitioners to the usage of Kaa. The notebook even contains a subset of the examples shown above, giving the auidence a chance to run the code for themselves. An immediate next step is to deploy extensive profiling to find performance bottlenecks and subsequently improve on them.

5 Conclusions

We present Kaa, a Python implementation of reachable set computation of nonlinear systems that is focussed towards accessibility and pedagocial use. The usage of inbuilt *sympy* libraries makes the implementation very short and simple (only 650 LOC). While we do incur performance drawbacks from selecting Python for implementing this algorithm, we be-

Table 3: Reachable Set Computation Time of Benchmarks

Model	**Kaa**	**SAPO** (C++)
SIR	11.41 sec	0.16 sec
Rossler	41.92 sec	1.17 sec
Quadcopter	78.21 sec	11.98 sec
Lotka-Volterra	18 min 95.05 sec	57.48 sec
Phosphoraley	103.81 sec	24.86 sec

lieve that it aids in fast prototyping and enables students to easily build on top of the library. In particular, Python's readability and extendibility will allow curious students to experiment with more involved bundle-transformations. The code can be accessed through https://github.com/Tarheel-Formal-Methods/kaa

References

[1] Dang, T., Dreossi, T., Piazza, C.: Parameter synthesis using parallelotopic enclosure and applications to epidemic models. In: International Workshop on Hybrid Systems Biology. pp. 67–82. Springer (2014)

[2] Dang, T., Salinas, D.: Image computation for polynomial dynamical systems using the bernstein expansion. In: International Conference on Computer Aided Verification. pp. 219–232. Springer (2009)

[3] Dang, T., Testylier, R.: Reachability analysis for polynomial dynamical systems using the bernstein expansion. Reliable Computing **17**(2), 128–152 (2012)

[4] Dreossi, T.: Sapo: Reachability computation and parameter synthesis of polynomial dynamical systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control. pp. 29–34 (2017)

[5] Dreossi, T., Dang, T., Piazza, C.: Parallelotope bundles for polynomial reachability. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control. pp. 297–306 (2016)

[6] Dreossi, T., Dang, T., Piazza, C.: Reachability computation for polynomial dynamical systems. Formal Methods in System Design **50**(1), 1–38 (2017)

[7] Garloff, J.: The bernstein expansion and its applications. Journal of the American Romanian Academy **25**, 27 (2003)

[8] Muñoz, C., Narkawicz, A.: Formalization of bernstein polynomials and applications to global optimization. Journal of Automated Reasoning **51**(2), 151–196 (2013)

[9] Nataraj, P.S., Arounassalame, M.: A new subdivision algorithm for the bernstein polynomial approach to global optimization. International journal of automation and computing **4**(4), 342–352 (2007)

[10] Nataray, P., Kotecha, K.: An algorithm for global optimization using the taylor–bernstein form as inclusion function. Journal of Global Optimization **24**(4), 417–436 (2002)

[11] Sassi, M.A.B., Testylier, R., Dang, T., Girard, A.: Reachability analysis of polynomial systems using linear programming relaxations. In: International Symposium on Automated Technology for Verification and Analysis. pp. 137–151. Springer (2012)

[12] Smith, A.P.: Fast construction of constant bound functions for sparse polynomials. Journal of Global Optimization **43**(2-3), 445–458 (2009)

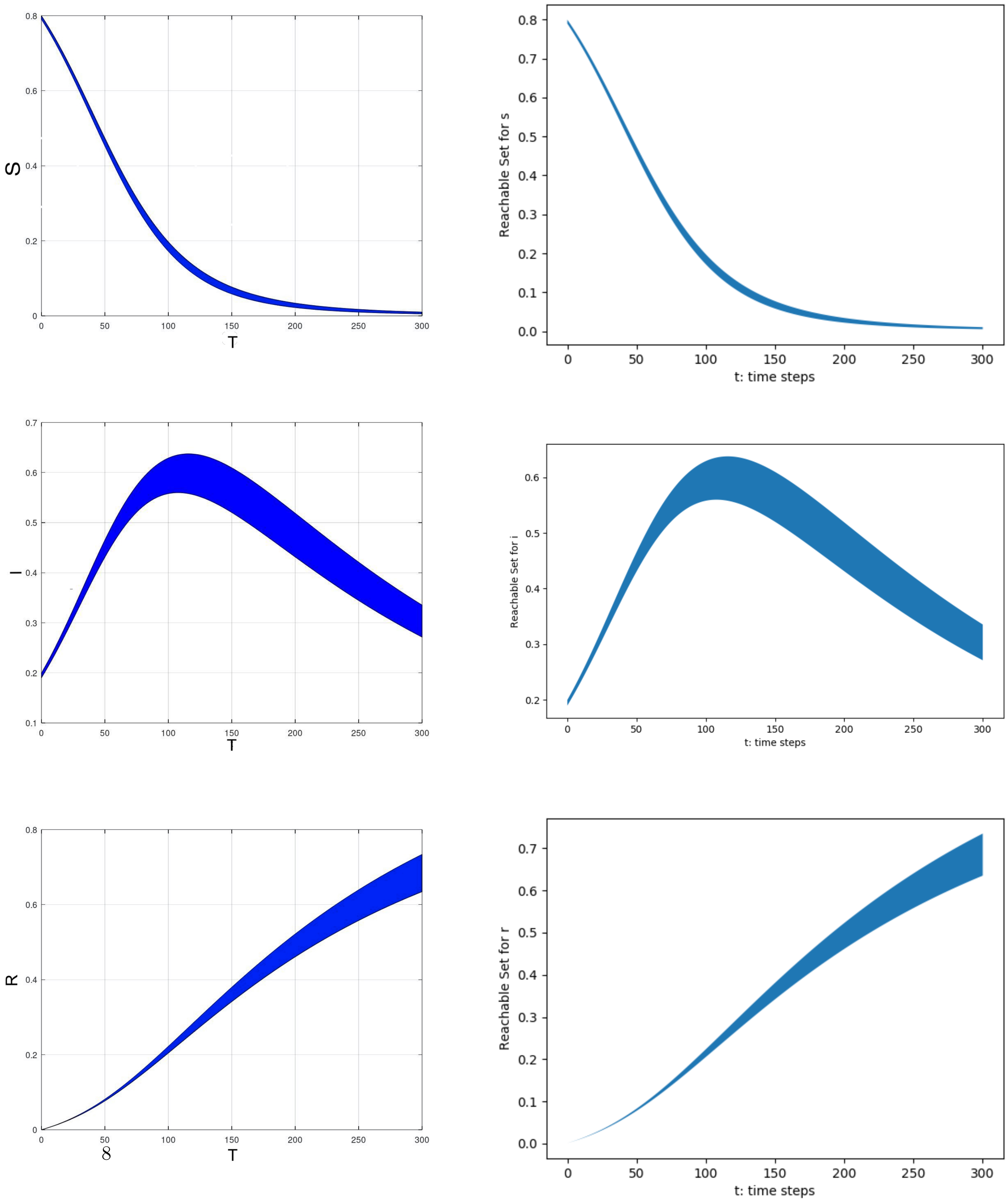

Figure 1: Figure depicting the reachable set computation of the SIR model.

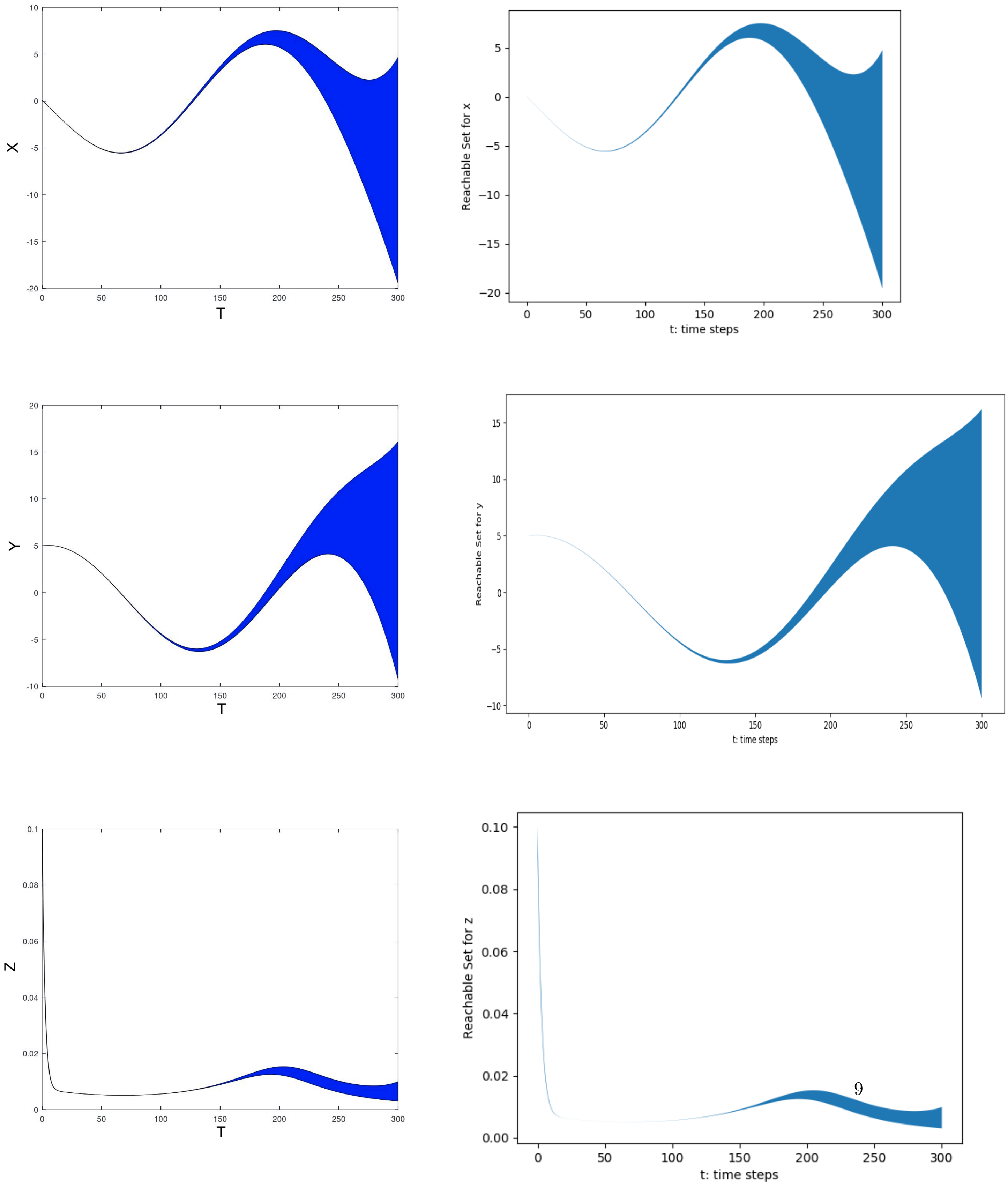

Figure 2: Figure depicting the reachable set computation of the Rossler model.

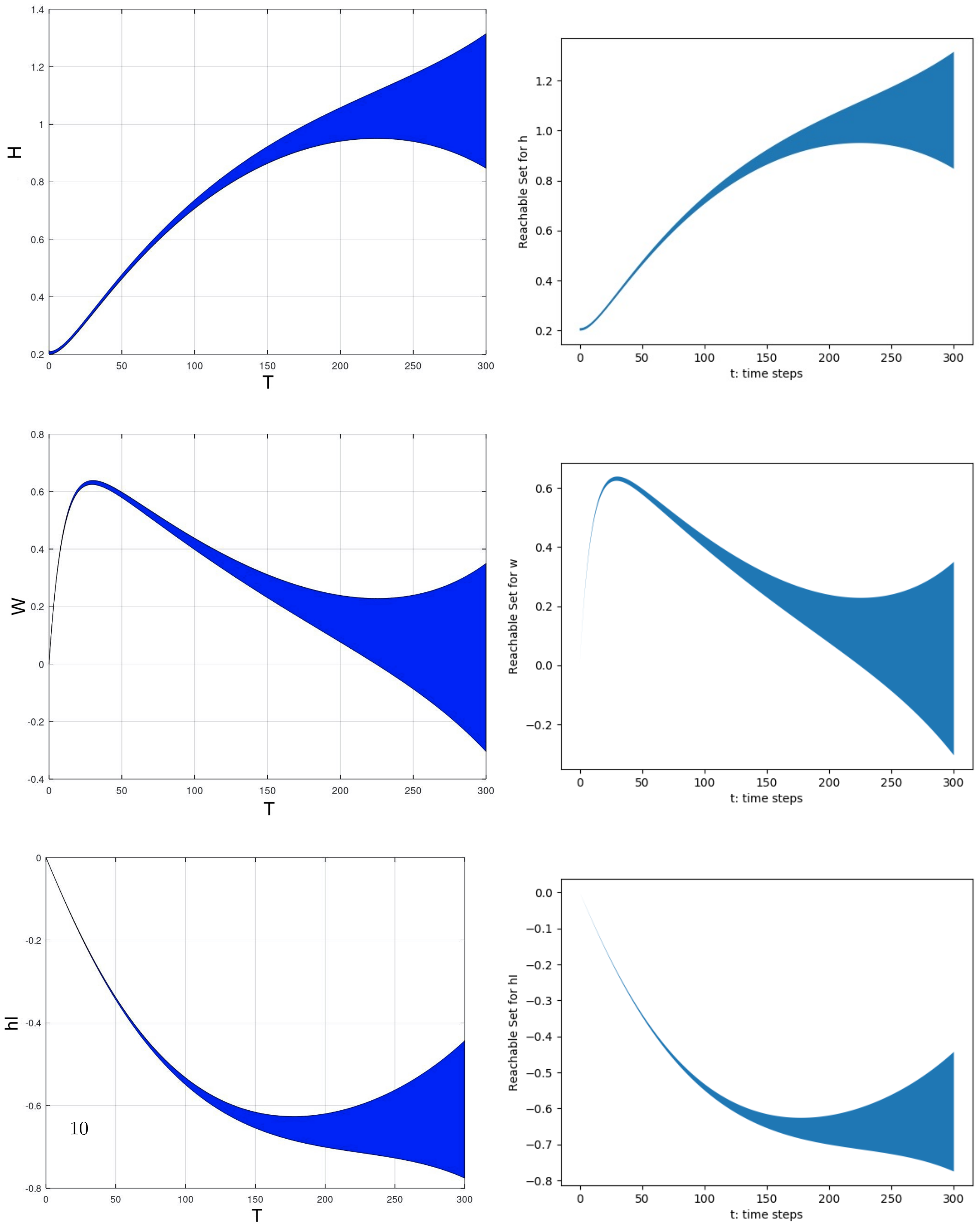

Figure 3: Figure depicting the reachable set computation of the Quadcopter model.

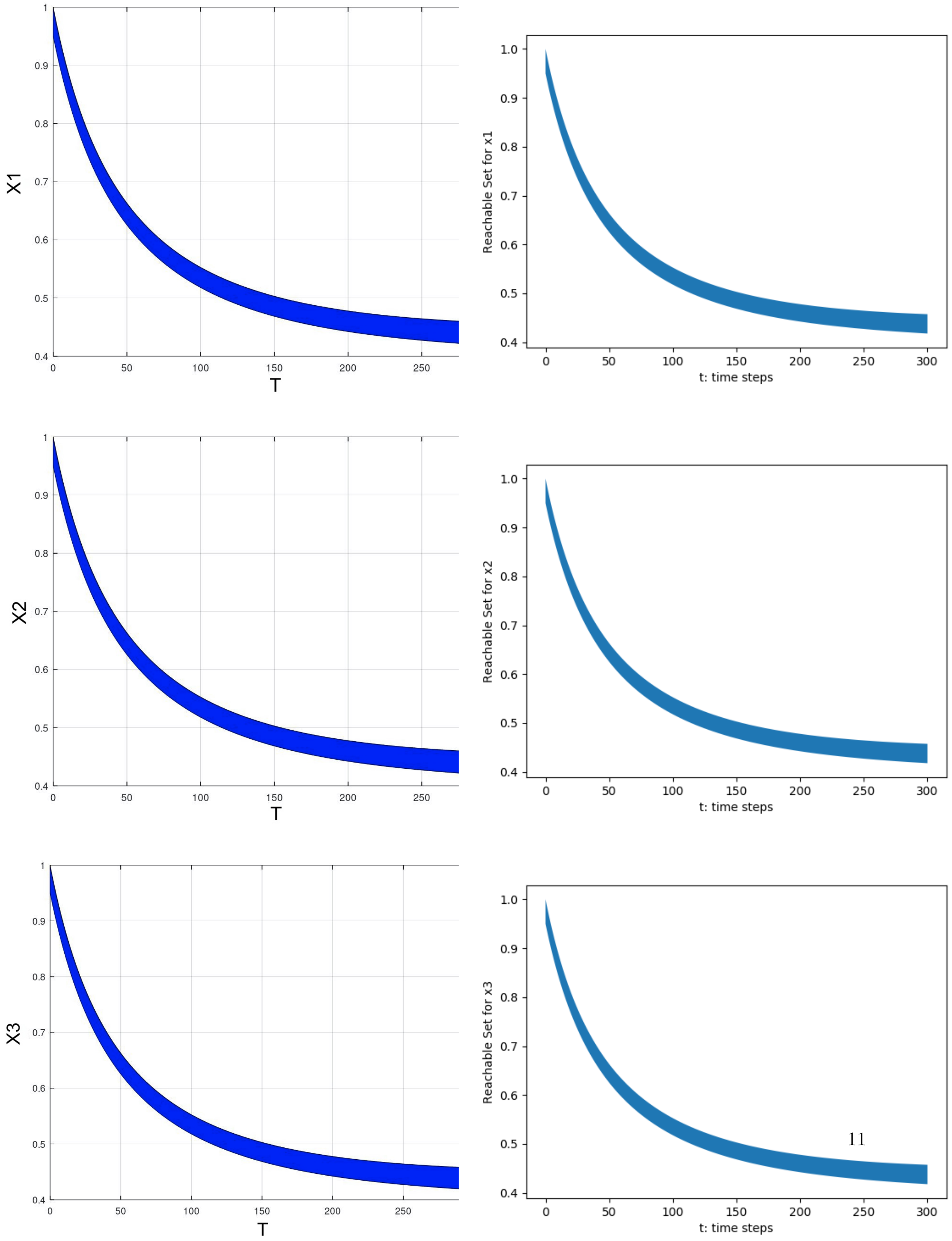

Figure 4: Figure depicting the reachable set computation of the Lotka-Volterra model.

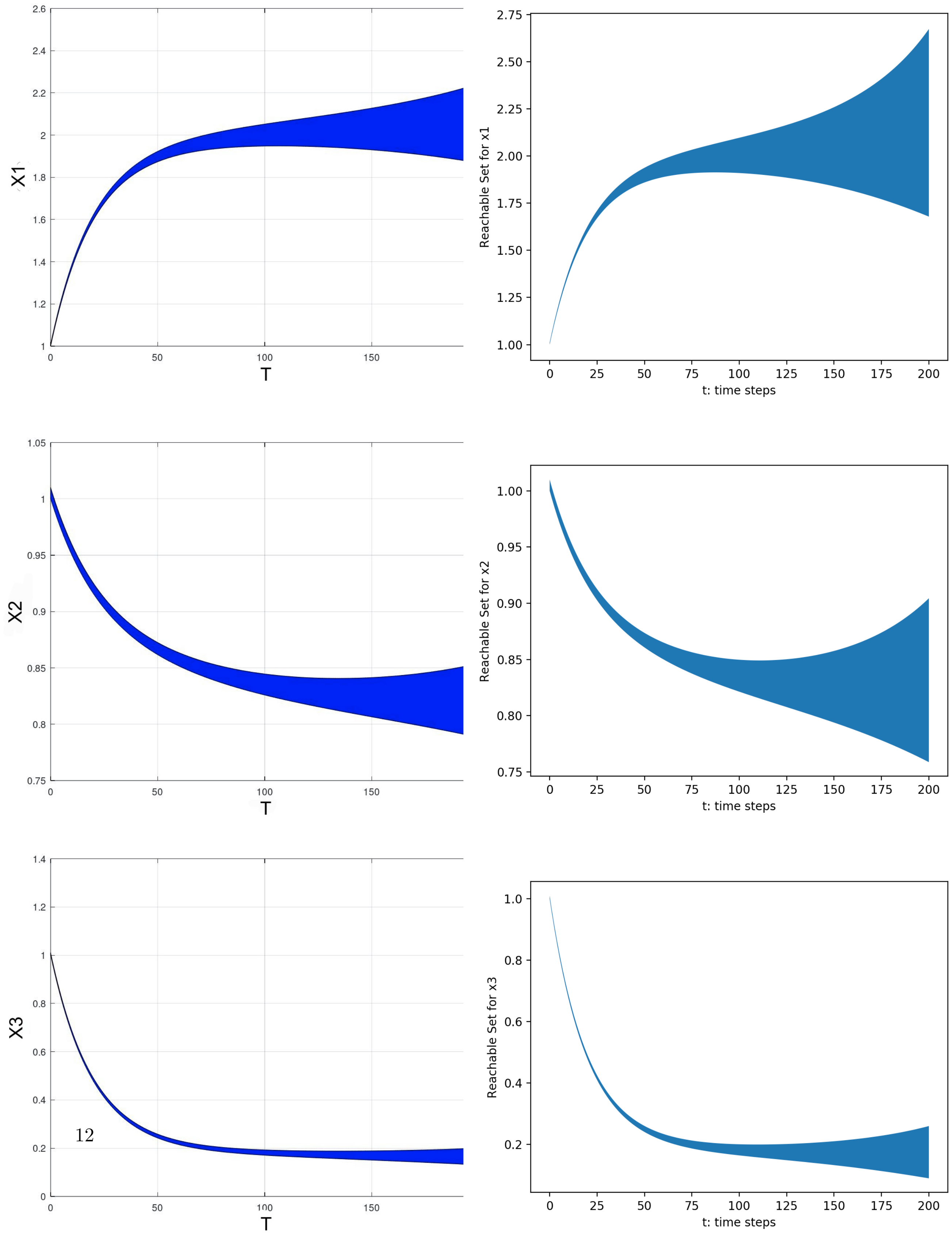

Figure 5: Figure depicting the reachable set computation of the Phosporley model.