Supplementary Information

Article:
bSTAB
An open-source software for computing the basin stability of multi-stable dynamical systems

Journal:
Nonlinear Dynamics

Authors:
Merten Stender, Hamburg University of Technology, m.stender@tuhh.de
Norbert Hoffmann, Hamburg University of Technology

bSTAB-M user manual

Merten Stender (m.stender@tuhh.de),
Hamburg University of Technology, Dynamics Group (www.tuhh.de/dyn)
https://github.com/TUHH-DYN/bSTAB/

January 22, 2021

Contents

1 Introduction

The basin stability concept relies on measuring the volumes of the basins of attraction $\mathcal{B}$ in the D− dimensional state space. The basin stability value $\mathcal{S}_{\mathcal{B}}$ (A) of the attractor A is given by [2, 1, 3]

$$\mathcal{S}_{\mathcal{B}}(A) = \int \kappa_{\mathcal{B}}(\mathbf{y})\, \rho(\mathbf{y})\, \mathrm{d}\mathbf{y}, \quad \mathbf{y} \in \mathbb{R}^D, \quad \mathcal{S}_{\mathcal{B}} \in [0, 1] \tag{1}$$

where $\kappa(\mathbf{y})$ is an indicator function

$$\kappa(\mathbf{y}) = \begin{cases} 1, & \text{if } \mathbf{y} \in \mathcal{B}(\mathrm{A}) \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

stating whether a trajectory converged to the attractor A, and thus whether **y** belongs to the basin $\mathcal{B}$ (A). ρ is a density distribution of states that the system may be perturbed to with $\int_{\mathbb{R}} \rho(\mathbf{y})\, \mathrm{d}\mathbf{y} = 1$. Estimates for the basin stability are obtained through Monte Carlo sampling from ρ, resulting in the absolute standard error of the estimate

$$e_{\text{abs}} = \sqrt{\mathcal{S}_{\mathcal{B}}(A)\,(1 - \mathcal{S}_{\mathcal{B}}(A))\,/N} \tag{3}$$

for the N-times repeated Bernoulli experiment. Hence, the choice of N is independent of the state space dimension, which is especially helpful for high-dimensional systems. The relative error of the estimate

$$e_{\text{rel}} = 1/\sqrt{N\mathcal{S}_{\mathcal{B}}(A)} \tag{4}$$

becomes relevant for small basin stability values, potentially requiring to increase N into the range N $\sim 1/\mathcal{S}_{\mathcal{B}}$ (A) [1]. Each state is numerically integrated, and the fraction of trajectories approaching the respective attractor is calculated to yield the basin stability estimate. In the following, we use $\mathcal{S}_{\mathcal{B}}$ to indicate the estimate of the actual basin stability value for reasons of readability.

The `bSTAB` toolbox aims at making the basin stability analysis easier to perform, hence helping this global stability metric make its way to a state-of-the-art analysis tool for nonlinear dynamical systems. `bSTAB` is designed to be model-agnostic, i.e. the user is required to have only basic knowledge about the system at hand, typical ODE formulations can be re-used, and only very limited coding skills are needed. Hence, this toolbox can be integrated into existing analysis routines without much effort. The toolbox is licensed under the GNU General Public License v3.0 and is freely available through `https://github.com/TUHH-DYN/bSTAB/`, which aims at providing easy access and high visibility. The code is under active development, and the community can add functionalities or report issues within the `git` framework. This project shall foster communication and collaborations across scientific disciplines, and at the same time grow the functionalities of `bSTAB` further.

2 Structure of the program

2.1 Folder structure

`bSTAB` is structured following a flat file and folder hierarchy. All built-in functions are contained in the top-level folder `./utils_bSTAB`. User cases will also be located on the top level folder, see the folder structure shown in Figure 1. The initialization function `init_bSTAB.m`, which sets all directories and initializes the `props` structure, must be located on the active search path. Each user case (e.g. `/case_pendulum`) must at least contain the ODE function definition file (e.g. `ode_pendulum.m`), the case definition file (e.g. `setup_pendulum.m`) and the feature extraction function (e.g. `features_pendulum.m`). The main script (e.g. `main_pendulum_case1.m`) runs the basin stability analysis. The user might be interested in multiple studies for the same system, hence subfolders for subcases (e.g. `case_1`) will be generated within the case folder. For example, different choices of the region of interest and different model or hyperparameter studies will result in several subcases and the corresponding folders. Within each subcase folder, all graphical output (`.fig` files) as well as the case setup (`props.mat`) and the results (`results.mat`) are stored.

```
./
├─ init_bSTAB.m
├─ utils_bSTAB/
│  ├─ compute_bs.m
│  ├─ compute_bs_ap.m
│  └─ ...
├─ case_pendulum/
│  ├─ main_pendulum_case1.m
│  ├─ setup_pendulum.m
│  ├─ ode_pendulum.m
│  ├─ features_pendulum.m
│  ├─ case_1/
│  │  ├─ fig_basinstability.fig
│  │  ├─ fig_statespace.fig
│  │  ├─ ...
│  │  ├─ props.mat
│  │  └─ results.mat
│  └─ case_2/
│     └─ ...
└─ case_duffing/
   └─ ...
```

Figure 1: Directory structure of the `bSTAB-M` toolbox

2.2 Case definition

The complete setup of a `bSTAB` case is defined in the setup file which serves as anchor for the complete computation: all model parameters and hyperparameters are defined at one place in `props`, which is a Matlab `structure array` that enables easy handling of many parameters. Table 1 lists all parameters that are required to be specified in `setup_system.m` for running an analysis.

General program settings		
`flagParallel`	`bool`	flag for enabling usage of parallel computing
`flagShowFigures`	`bool`	flag for showing figures during the computation
`flagUseHPC`	`bool`	flag for suppressing all GUI outputs
`progessBar`	`bool`	flag for enabling a GUI progress bar
Dynamical system: `.model` structure array		
`odeFun`	`function handle`	the ODE definition
`dof`	`int`	number of states D
`odeParams`	`array`	model parameters **p** (default values)
Region of interest: `.roi` structure array		
`N`	`int`	number of samples N
`minLimits`	`array`	minimum values per state space dimension
`maxLimits`	`array`	maximum values per state space dimension
`samplingVarDims`	`array of bool`	flag indicating which states to sample from
`samplingPDF`	`string`	sampling probability density function ρ
`samplingCustomFun`	`function handle`	custom PDF definition ρ (optional)
Time integration parameters: `.ti` structure array		
`fs`	`float`	sampling rate f_s
`tSpanStart`	`array`	integration start time
`tSpanEnd`	`array`	integration end time
`tstar`	`float`	steady-state time (no transients after t^*)
`timeStepper`	`string`	time stepping routine
`options`	`struct`	time stepper options using Matlab's `odeset()`
Clustering options: `.clust` structure array		
`clustMod`	`string`	clustering mode (supervised / unsupervised)
`featExtractFun`	`function handle`	feature extraction function ϕ
`numFeatures`	`int`	number of features (length of **X**)
`clustMethod`	`string`	clustering algorithm (kNN, ...)
`clustMethodTol`	`float`	additional parameter for the clustering method
Solution templates (*only required for* `clustMode=='supervised'`): `.templ` structure array		
`k`	`int`	number of solutions k
`Y0`	`cell array`	initial state leading to the template solutions
`modelParams`	`cell array`	corresponding model parameter values
`label`	`cell array`	string labeling the solution template
Evaluation parameters: `.eval` structure array		
`ampFun`	`function handle`	amplitude computation function

Table 1: Structure of the setup file defining a case in `bSTAB`: fields of the `props` structure (left column), variable types (middle column) and descriptions (right column)

2.3 Custom function definitions

Most fields of the case definition in `props` should be self-explanatory following the presentation in the introduction and in the original scientific work. However, several `function handles` must be specified, which requires minimal coding by the user of the toolbox. The following paragraphs illustrate how to implement those system-specific functions. Particularly, the examples are taken from the damped driven pendulum case that is provided in `bSTAB-M` in the `./case_pendulum/` directory.

2.4 Case definition function

Setting up the case requires specifying all relevant parameters from Table 1 in a setup file that will be called by the main script. Following, the case definition file `setup_pendulum.m` is given for the pendulum case.

```
function  [props] = setup_pendulum(props)

%% 0. general program settings
props.flagParallel = true;
props.flagShowFigures = true;
props.flagUseHPC = false;
props.progessBar = true;

%% 1. dynamical system properties (.model struct)
props.model.odeFun = @ode_pendulum; % a function handle @my_ode_function
props.model.dof = 2; % degrees-of-freedom (= length of state vector)
alpha = 0.1;     % p1
T = 0.5;         % p2
K = 1.0;           % p3
props.model.odeParams = [alpha, T, K];

%% 2. Region of Interest (.roi struct)
props.roi.N = 10000; % integer number.
props.roi.minLimits = [-pi+asin(T/K), -10]; % must be of length <props.model.dof>
props.roi.maxLimits = [pi+asin(T/K), 10]; % must be of length <props.model.dof>
props.bs.samplingVarDims = [true; true]; % boolean: dims to vary
props.roi.samplingPDF = 'uniform';
props.bs.samplingCustomFun = ''; % a function handle @

%% 3. Time integration parameters (.ti struct)
props.ti.fs = 25; %sampling frequency fs = 1/dt
props.ti.tSpanStart = 0;
props.ti.tSpanEnd = 1000;
props.ti.tSpan = [props.ti.tSpanStart props.ti.tSpanEnd];
props.ti.tStar = props.ti.tSpan(end)-50;
props.ti.timeStepper = 'ode45';
options = odeset('RelTol',1e-8);
props.ti.options = options;

%% 4. Clustering options (.clust struct)
props.clust.clustMode = 'supervised'; % string
props.clust.featExtractFun = @features_pendulum; % a function handle
props.clust.numFeatures = 2;
props.clust.clustMethod = 'kNN'; % default: kNN(k=1) using Euclidean distance
props.clust.clustMethodNorm = 'euclidean';
props.clust.clustMethodTol = NaN;

%% 5. Templates (for the supervised clustering setting)
props.templ.k = 2; %number of different steady-state solutions
props.templ.Y0{1} = [0.5; 0];    % initial condition (NOT the steady-state itself)
props.templ.modelParams{1} = [0.1, 0.5, 1.0]; % model parameters
props.templ.label{1} = 'FP';     % stable fixed point label
props.templ.Y0{2} = [2.7; 0];
```

```
props.templ.modelParams{2} = [0.1, 0.5, 1.0];
props.templ.label{2} = 'LC'; % limit cycle solution label

%% 6. Evaluation
props.eval.ampFun = @extract_amps; % a function handle. Default: @extract_amps

%% 7. Bug check
props = check_props(props);

%% store the parameters locally to the project subfolder
save([props.subCasePath, '/props.mat'], 'props');

end
```

2.4.1 ODE function definition

The definition of the dynamical system in `props.model.odeFun` fully aligns with the classical formulation in `Matlab` for the `ode` solvers. For example, the ODE definition for the damped driven pendulum reads

```
function [dydt] = ode_pendulum(t, y, alpha, T, K)

dydt = [y(2);
        -alpha*y(2)+T-K*sin(y(1))];

end
```

The function takes time `t`, the state vector `y` and model parameters `alpha, T, K` as inputs and returns the vector of time-derivatives `dydt`. A simple time integration for $\alpha = 0.1$, $T = 0.5$, $K = 1$ for the time $t = 0, \ldots, 100$ starting from the initial condition $\mathbf{y}_0 = [0, 0]^\top$ can then be performed by

```
[T,Y] = ode45(@(t,y) ode_pendulum(t,y,0.1,0.5,1.0), [0,100], [0;0])
```

2.4.2 Feature extraction function definition

The feature extraction function $\mathbf{X} = \phi(\mathbf{y}(t))$ is individual to the system at hand, and domain knowledge is beneficial for defining a minimal set of features that enable the classification. The overall structure of the feature extraction function in `props.clust.featExtractFun` requires the time vector `T`, the trajectory matrix `Y`, and the `props` structure as input. The function must return a column vector of features `X` as output. The function

```
function [X] = features_pendulum(T, Y, props)

% 1. detect the steady-state regime (time after props.ti.tStar)
idx_steady = find(T>props.ti.tStar,1);

% 2. extract some features (must work for all values of T!)
Delta = abs(max(Y(idx_steady:end,2)) - mean(Y(idx_steady:end,2)));

% one-hot encoded labels
if Delta<0.01
    X(1,1) = 1; %FP
    X(2,1) = 0;
else
    X(1,1) = 0; %LC
    X(2,1) = 1;
end
```

```
end
```

extracts two features from the steady-state trajectories ($t > t^*$) according to the one-hot encoded thresholded deviation of the maximum rotational velocity (second state) about the average rotational velocity following

$$\mathbf{X} = [X_1, X_2]^\top = \begin{cases} [1,0]^\top, & \text{if } \delta \leq 0.01 \\ [0,1]^\top, & \text{otherwise} \end{cases}, \quad \delta = |\max(\tilde{y}_2) - \text{mean}(\tilde{y}_2)| \quad . \tag{5}$$

2.4.3 Custom probability density function definition

The user can specify custom density functions $\rho(\mathbf{y})$ by a function handle in `props.roi.samplingCustomFun`. For example, custom density functions can be used to sample from experimentally observed distributions. The function must return a $N \times D$ array of states in `IC`, and must accept the number of samples `n_points`, the minimal `min_vals` and maximum `max_vals` value ranges per state space dimension as well as a boolean array `var_dims` indicating which state space dimension to consider for the sampling:

```
function [IC] = customDensityFun(n_points, min_vals, max_vals, var_dims)
```

2.4.4 Amplitude function definition

When model parameter studies are run in `bSTAB`, there is the possibility to extract amplitude information from each of the N time integration results. The amplitude information can then be used to generate a bifurcation diagram, i.e. display the amplitude against the model parameter variation. Per default, the maximum absolute value per state in the steady-state regime $\max(\mathbf{y}(t > t^*))$ is computed by the toolbox, as implemented in `extract_amps.m`. Taking the time integration results `T`, `Y` and the `props` structure as input, a custom amplitude extraction function can return a vector of amplitudes in `amps`:

```
function [amps] = extract_amps(T, Y, props)
```

3 Tutorial case studies

3.1 Damped driven pendulum

Following the original work by Menck et al. [2] we replicate the results using `bSTAB-M`. The ODE definition and the feature extraction functions were already given in the previous section. For completeness, the case definition file is given below. If not stated differently in the upcoming paragraph, this configuration is employed for the analysis.

```
function  [props] = setup_pendulum(props)

%% 0. general program settings
props.flagParallel = true;
props.flagShowFigures = true;
props.flagUseHPC = false;
props.progessBar = true;

%% 1. dynamical system properties (.model struct)
props.model.odeFun = @ode_pendulum; % a function handle @my_ode_function
props.model.dof = 2; % degrees-of-freedom (= length of state vector)
alpha = 0.1;    % p1
```

```
T = 0.5;        % p2
K = 1.0;          % p3
props.model.odeParams = [alpha, T, K];

%% 2. Region of Interest (.roi struct)
props.roi.N = 10000; % integer number.
props.roi.minLimits = [-pi+asin(T/K), -10]; % must be of length <props.model.dof>
props.roi.maxLimits = [pi+asin(T/K), 10]; % must be of length <props.model.dof>
props.bs.samplingVarDims = [true; true]; % boolean: dims to vary
props.roi.samplingPDF = 'uniform';
props.bs.samplingCustomFun = ''; % a function handle @

%% 3. Time integration parameters (.ti struct)
props.ti.fs = 25; %sampling frequency fs = 1/dt
props.ti.tSpanStart = 0;
props.ti.tSpanEnd = 1000;
props.ti.tSpan = [props.ti.tSpanStart props.ti.tSpanEnd];
props.ti.tStar = props.ti.tSpan(end)-50;
props.ti.timeStepper = 'ode45';
options = odeset('RelTol',1e-8);
props.ti.options = options;

%% 4. Clustering options (.clust struct)
props.clust.clustMode = 'supervised'; % string
props.clust.featExtractFun = @features_pendulum; % a function handle
props.clust.numFeatures = 2;
props.clust.clustMethod = 'kNN'; % default: kNN(k=1) using Euclidean distance
props.clust.clustMethodNorm = 'euclidean';
props.clust.clustMethodTol = NaN;

%% 5. Templates (for the supervised clustering setting)
props.templ.k = 2; %number of different steady-state solutions
props.templ.Y0{1} = [0.5; 0];   % initial condition (NOT the steady-state itself)
props.templ.modelParams{1} = [0.1, 0.5, 1.0]; % model parameters
props.templ.label{1} = 'FP';    % stable fixed point label
props.templ.Y0{2} = [2.7; 0];
props.templ.modelParams{2} = [0.1, 0.5, 1.0];
props.templ.label{2} = 'LC'; % limit cycle solution label

%% 6. Evaluation
props.eval.ampFun = @extract_amps; % a function handle. Default: @extract_amps

%% 7. Bug check
props = check_props(props);

%% store the parameters locally to the project subfolder
save([props.subCasePath, '/props.mat'], 'props');

end
```

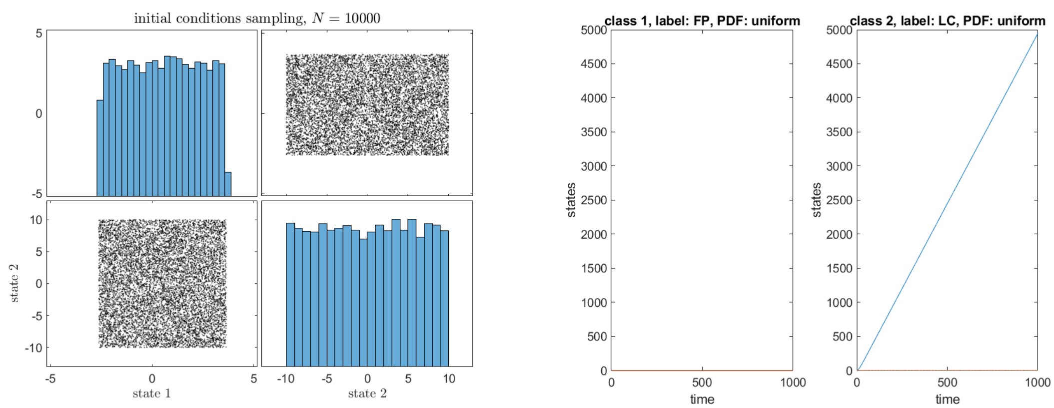
3.1.1 Basin stability analysis at fixed model parameters

The basin stability for the bi-stable configuration at T = 0.5 is obtained by running `compute_bs.m`. The main script (`main_pendulum_case1.m`) is defined as follows:

```
clear; close all; clc;

% define a name for the current analysis
currentCase = 'publication_case_1';

% set up paths, initialize bSTAB, create properties struct <props>
[props] = init_bSTAB(currentCase);
```

```
%% 1. set up your case
[props] = setup_pendulum(props);

%% 2. compute the basin stability
[res_tab, res_detail, props] = compute_bs(props);

% save the results
save([props.subCasePath, '/results_basinstability.mat']);
```

The current analysis is named `case_1`, which will create a new sub-directory in `./case_pendulum/case_1/` where all results and figures will be stored. `bSTAB` is initialized through `init_bSTAB`. The `props` structure is populated in the case definition `setup_pendulum` (see previous section) and the basin stability of both solutions is return by `compute_bs`. Several plots can be generated during the computation. First, the sampling points and their distribution is returned in a figure stored to `fic_ic_sampling`. Secondly, if the clustering mode is `supervised`, the template solutions are stored to the figure `fig_solution_templates`. Figure 2 depicts the graphical output of these two figures.



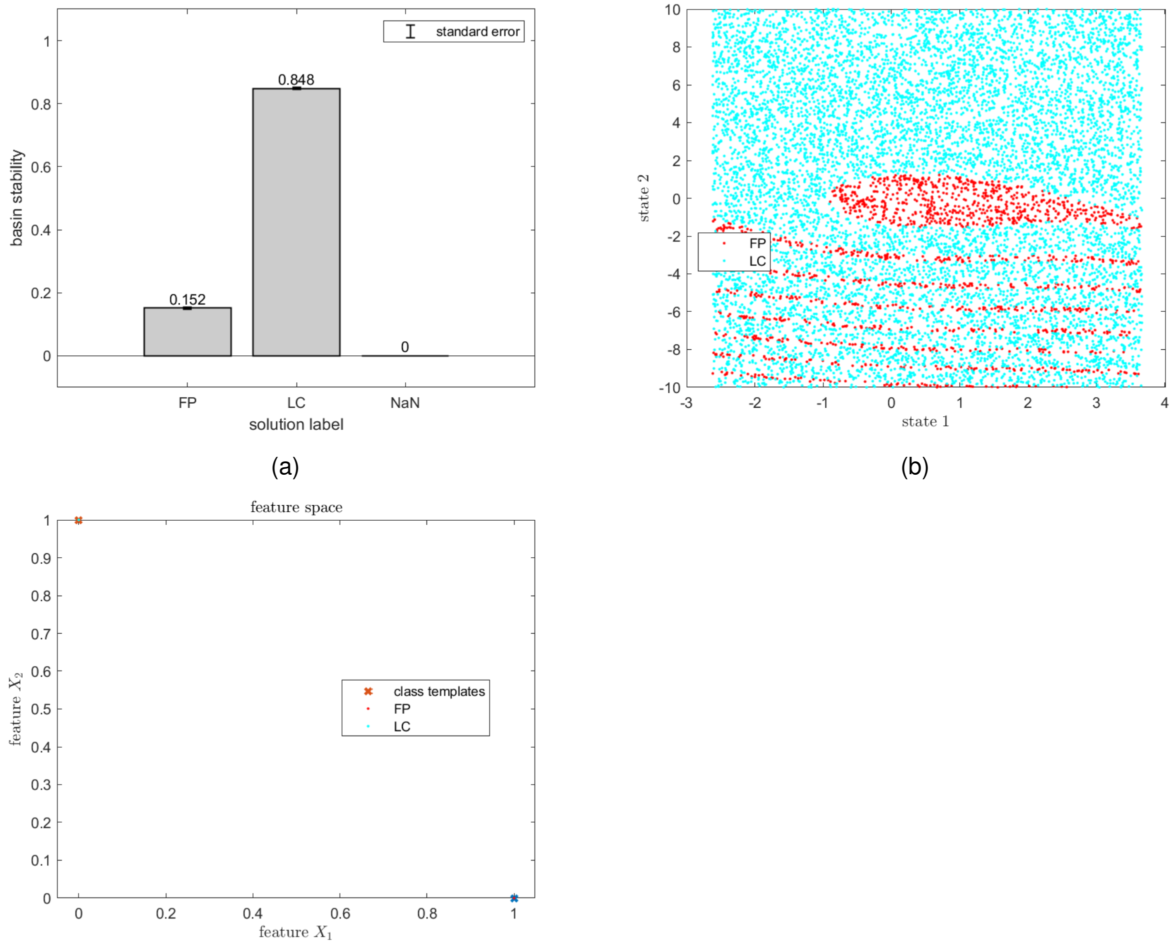
(a) (b)

Figure 2: Graphical output during the basin stability computation for the pendulum case: (a) distribution of sampling points, (b) templates for the solutions provided by the user

The results obtained by the basin stability analysis can be visualized in several ways using the following commands:

```
% 1. bar plot for the basin stability values
plot_bs_bargraph(props, res_tab, true);

% 2. state space scatter plot: class-labeled initial conditions
plot_bs_statespace(props, res_detail, 1, 2);

% 3. feature space and classifier results
plot_bs_featurespace(props, res_detail);
```

First, the basin stability values are displayed in the form of a bar graph, see Figure 3 (a). Besides the two solution labels provided by the user in the case definition file, a third solution `NaN` is shown. `bSTAB-M` provides this class for sample points which cannot by classified to belong to either of the template vectors. Hence, non-zero entries for the `NaN` class indicate issues in the classification or clustering task, which most often can be traced back to the feature extraction function or to the clustering algorithm options. Secondly, the state space is shown with states colored by their class label, see Figure 3 (b). Thirdly, the feature space is shown in Figure 3 (c).



(a) (b) (c)

Figure 3: Graphical output after the basin stability computation for the pendulum case: (a) basin stability values, (b) state space and (c) feature space

3.1.2 Basin stability analysis along model parameter

To study the evolution of the basin stability values along a model parameter, i.e. in analogy to bifurcation diagrams, the following main function is employed. Particularly, a variation of the torque `T` is studied in `main_pendulum_case2.m`

```
% ensure a clean start
clear; close all; clc;

% define a name for the current analysis
currentCase = 'publication_case2';

% set up paths, initialize bSTAB, create properties struct <props>
[props] = init_bSTAB(currentCase);


%% 1. set up your case
```

```
[props] = setup_pendulum(props);


%% 2. compute basin stability along model parameter variation
% Specify that you want to vary a model parameter
props.ap_study.mode = 'model_parameter';

% identify the props struct element that you want to vary.
props.ap_study.ap = 2;

% specify the parameter variation vector
props.ap_study.ap_values = 0.01:0.05:1.0;

% specify the name of the adaptive parameter (just for plotting purpose)
props.ap_study.ap_name = '$T$';

% let bSTAB compute basin stability sensitivity
[res_tab, res_detail, props] = compute_bs_ap(props);

% save the results
save([props.subCasePath, '/results.mat']);
```

This script will change the second ODE model parameter (`props.ap_study.ap = 2`) along the grid of torque values specified by the user in`props.ap_study.ap_values = 0.01:0.05:1.0`. Graphical analysis of the results can be obtained by calling

```
plot_bs_parameter_study(props, res_tab, false);
```

For the current case, Figure 4 depicts the evolution of the basin stability values along the model parameter as it is provided by `bSTAB`.

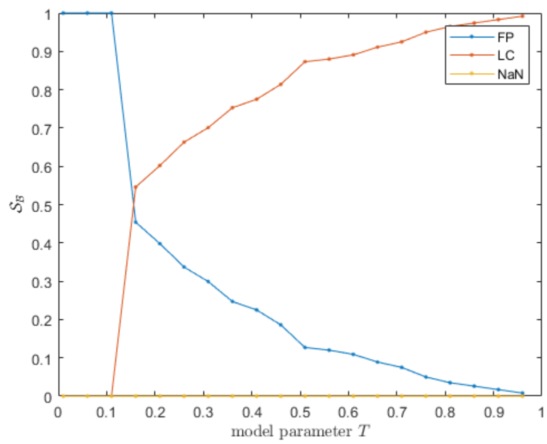


Figure 4: Basin stability values as a function of the model parameter T

References

[1] MENCK, P. J., HEITZIG, J., KURTHS, J., AND JOACHIM SCHELLNHUBER, H. How dead ends undermine power grid stability. *Nature communications 5* (2014), 3969.

[2] MENCK, P. J., HEITZIG, J., MARWAN, N., AND KURTHS, J. How basin stability complements the linear-stability paradigm. *Nature Physics 9*, 2 (2013), 89–92.

[3] MITRA, C., CHOUDHARY, A., SINHA, S., KURTHS, J., AND DONNER, R. V. Multiple-node basin stability in complex dynamical networks. *Physical review. E 95*, 3-1 (2017), 032317.