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/

March 19, 2021

Contents

1 Introduction

Many nonlinear dynamical systems exhibit **multi-stability**: at a fixed parameter configuration, the system $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \mathbf{y} \in \mathbb{R}^D$ has more than one possible steady-state solutions $\bar{\mathbf{y}}$. The time-asymptotic behavior hence solely depends on the initial condition $\mathbf{y}_0$ or instantaneous perturbations from which the trajectories $\mathbf{y}(t)$ evolve. Conceptually, the sensitivity of the steady-state behavior is prescribed by the shape of the basins of attraction $\mathcal{B}$. Classical local stability metrics (perturbation-based linearization) assess the stability of a state against *small perturbations*, i.e. stating whether the system will return back to the same attractor A after being perturbed. However, the size of small perturbations is not strictly defined. Moreover, even small perturbations may result in a different steady-state solution if the system exhibits multi-stability and the trajectories start close-by a basin boundary.

The concept of **basin stability** [1] introduces a global stability metric including *small* and *large* perturbations into the stability analysis by measuring the volumetric share of a basin of attraction within the state space. Mathematically, the basin stability value of the attractor A is defined as

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

κ is an indicator function (1 if $\mathbf{y} \in \mathcal{B}(A)$, 0 otherwise), and ρ is the density distribution of initial states $\int_{\mathbb{R}} \rho(\mathbf{y})\, \mathrm{d}\mathbf{y} = 1$. The basin stability values are estimated through sampling ρ by N Monte Carlo samples from a region of interest $\mathcal{Q} \subset \mathbb{R}^D$, e.g. the set of possible initial states. The absolute standard error hence is given by

$$e_{\text{abs}} = \sqrt{\mathcal{S}_{\mathcal{B}}(A)\left(1 - \mathcal{S}_{\mathcal{B}}(A)\right)/N} \quad , \tag{2}$$

and the relative error of the estimate is

$$e_{\text{rel}} = 1/\sqrt{N \mathcal{S}_{\mathcal{B}}(A)} \quad . \tag{3}$$

While the theoretical concept is rather simplistic, the implementation of a consistent and robust basin stability computation is not straight-forward. Furthermore, the practical basin stability computation involves several hyperparameters, which affect the results strongly if not chosen correctly. The sensitivity of the basin stability values against those hyperparameters calls for a highly automated computing pipeline that enables the user to perform grid-based searching for optimal hyperparameter selection.

This toolbox `bSTAB` introduces a programming framework for the computation of basin stability values for time-continuous nonlinear dynamical systems. The toolbox equips the user with easy-to-use routines that require only minimal inputs and coding. The dynamics expert must set up the computation case by specifying the system-related settings, such as the range of initial conditions to be studied, system parameter values, and template solutions for each of the a-priori known multi-stable solutions. Hereafter, the toolbox allows to compute the basin stability values by a total of 3 lines of code. Hyperparameter studies and variations of the basin stability along model parameter variations can be run in the same fashion without requiring any additional coding.

`bSTAB` shall foster collaborative research and interdisciplinary communication in the vast field of nonlinear dynamics. We aim at equipping researchers and practitioners with a ready-to-use code, that will help to integrate the basin stability analysis into the toolbox of stability analysis. The open-source code concepts will help to grow the functionalities of bSTAB, integrate more use cases and potentially migrate the current implementation to other, more efficient, programming languages. `bSTAB` is licensed under the GNU General Public License v3.0 and is freely available at https://github.com/TUHH-DYN/bSTAB/

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.

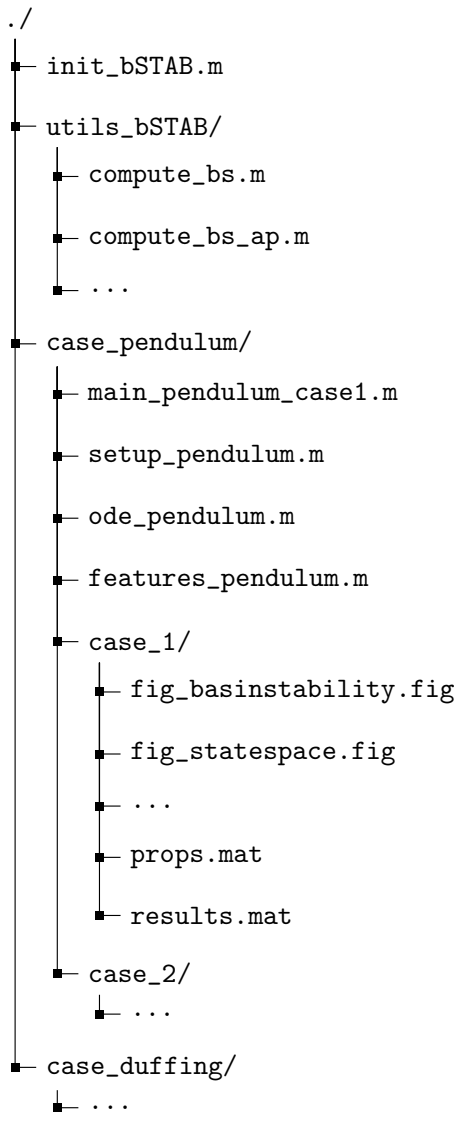


Figure 1: Directory structure of the bSTAB-M toolbox

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 α = 0.1, T = 0.5, K = 1 for the time t = 0, ... , 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

$$\mathbf{X} = [X_1, X_2]^\top = \begin{cases} [1,0]^\top, & \text{if } \delta \leq 0.01 \\ [0,1]^\top, & \text{otherwise} \end{cases}, \quad \delta = |\max(\tilde{y}_2) - \text{mean}(\tilde{y}_2)| \quad . \tag{4}$$

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. [1] 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
```

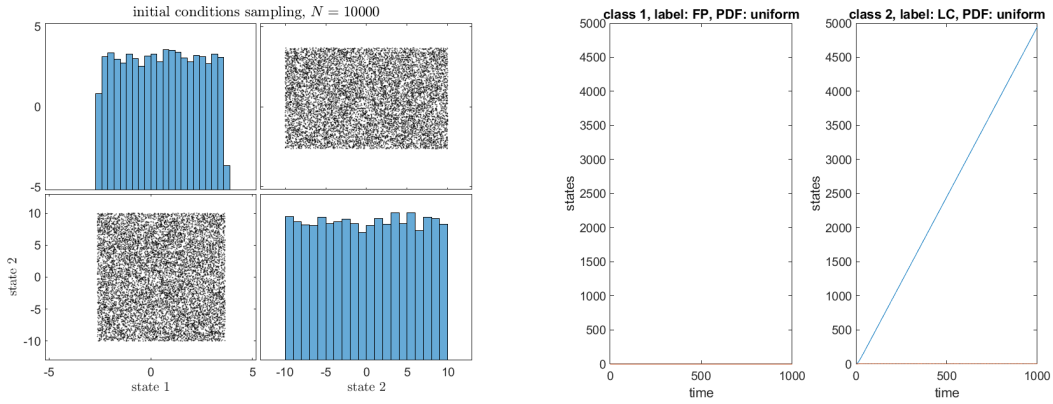
3.1.1 Basin stability analysis at fixed model parameters

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).

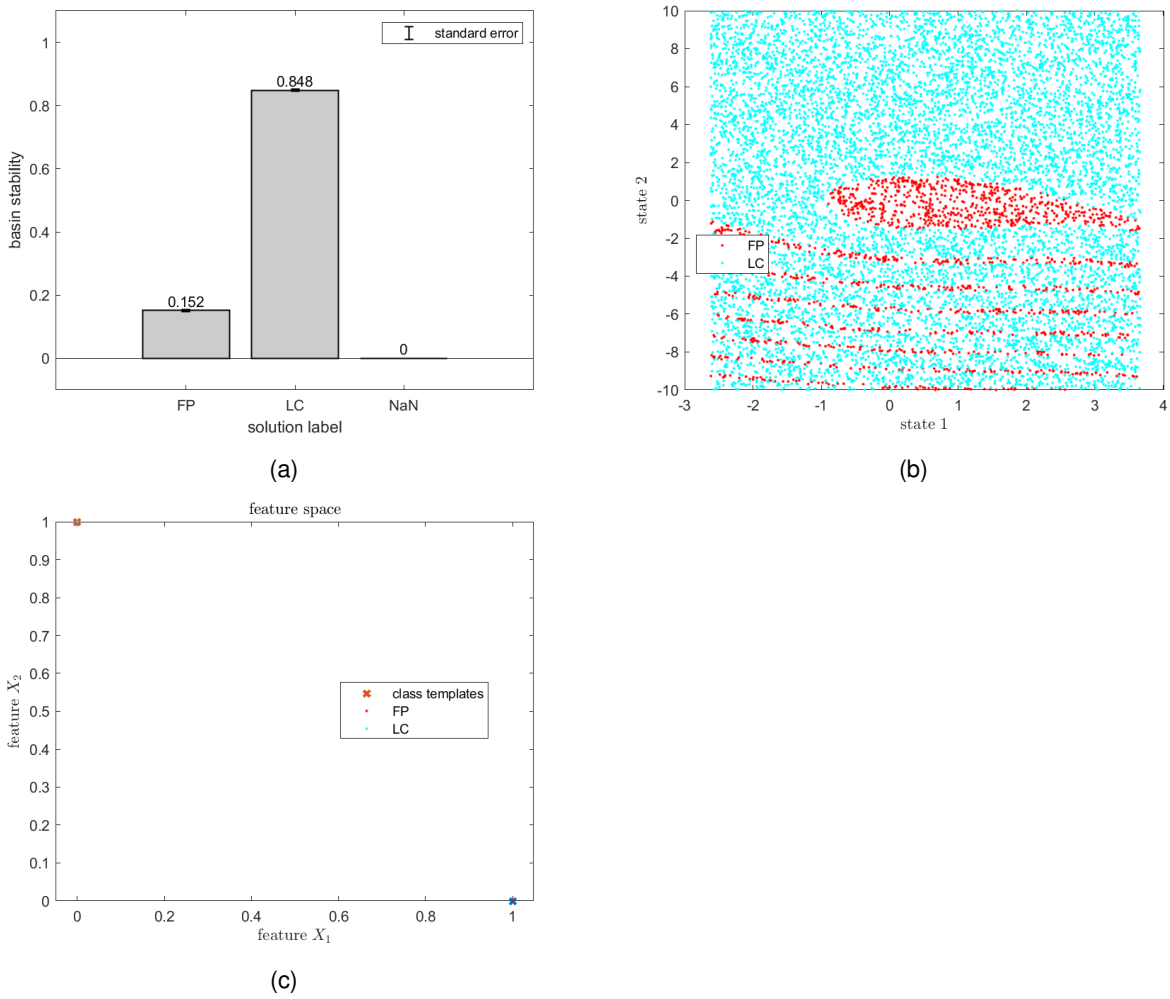


Figure 3: Graphical output after the basin stability computation for the pendulum case: (a) basin stability values, (b) state space and (c) feature space

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`.

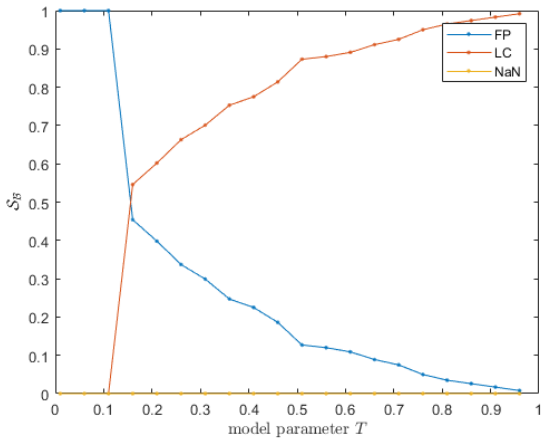


Figure 4: Basin stability values as a function of the model parameter T

3.2 Multi-stable Duffing oscillator

The much-studied Duffing oscillator

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= -\delta y_2 - k_3 y_1^3 + F\cos(t) \end{aligned} \tag{5}$$

exhibits interesting multi-stability features for specific parameter settings. Following Thompson and Stewart [2], setting the parameters $\delta = 0.08$, $k_3 = 1$, $F = 0.2$ results in a 5-fold multi-stable scenario: five limit cycle solutions arise (two period-1 cycles, two period-2 cycles, and one period-3 cycle).

Despite knowing about the specific multi-stable solutions, this system is a good candidate for showcasing the *unsupervised* clustering mode of `bSTAB`. In this mode, the user does not provide templates, i.e. initial conditions, as there is no a-priori knowledge about i) *how many* and ii) *which* solutions one can expect to encounter. `bSTAB` will run the Monte Carlo time integrations, extract features from the steady-state, and then come up with an optimal clustering in the feature space. The unsupervised clustering technique DBSCAN is used here, as (in contrast to classical clustering methods) this method does not require a-priori knowledge about how many clusters to form. Still, DBSCAN requires setting the minimal number of points required to form a cluster (default 10), and an epsilon distance which `bSTAB` sets itself for optimally separated clusters. Some specific situations may however require manual fine-tuning these parameters.

The ODE definition reads

```
function [dydt] = ode_duffing(t, y, delt, k3, A)

dydt = [ y(2);
    -delt*y(2)-k3*y(1)^3+A*cos(t)];

end
```

and for the feature extraction, simply the maximum and the standard deviation of the first state are extracted from the steady-state. Note that if no a-priori knowledge about the types of dynamics is available, the feature extraction becomes the most important key to differentiate different solutions. Hence, extra care must be taken to define the most discriminative and descriptive features here.

```
function [features] = extract_features_duffing(T, Y, props)

idx_steady = find(T>props.ti.tStar,1);
Y = Y(idx_steady:end,:);

features(1,1) = max(Y(:,1));
features(2,1) = std(Y(:,1));

end
```

The complete case setup is given below. Again, note that no template properties are required from the user, treating the dynamical system and its behavior as a black-box.

```
function  [props] = setup_duffing(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_duffing;
props.model.dof = 2;

delt = 0.08;    % p1
k3 = 1.0;        % p2
A = 0.2;         % p3
props.model.odeParams = [delt, k3, A];


%% 2. Region of Interest (.roi struct)
props.roi.N = 2000;

props.roi.minLimits = [-1.0, -0.5];
props.roi.maxLimits = [1.0, 1.0];
props.bs.samplingVarDims = [true; true];

props.roi.samplingPDF = 'uniform';
props.bs.samplingCustomFun = '';

%% 3. Time integration parameters (.ti struct)
props.ti.fs = 50;
props.ti.tSpanStart = 0;
props.ti.tSpanEnd = 1000;
props.ti.tSpan = [props.ti.tSpanStart props.ti.tSpanEnd];
props.ti.tStar = props.ti.tSpan(end)-100;

props.ti.timeStepper = 'ode45';
options = odeset('RelTol',1e-8);
props.ti.options = options;

%% 4. Clustering options (.clust struct)
props.clust.clustMode = 'unsupervised';

props.clust.featExtractFun = @features_duffing;
props.clust.numFeatures = 2;

%% 5. Templates (for the supervised clustering setting)
% ---> we're using the unsupervised setting, nothing to set here!

%% 6. Evaluation
props.eval.ampFun = @extract_amp_duffing;

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

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

3.2.1 Unsupervised basin stability computation

The basin stability computation in the *unsupervised* mode follows the same logic as for the *supervised* mode described in the previous pendulum example, i.e. through running `compute_bs.m`. The main script (`main_duffing_unsupervised.m`) is defined as follows:

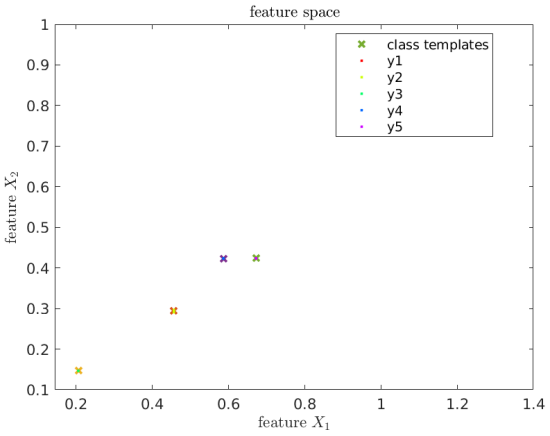
```
clear; close all; clc; addpath('..');

currentCase = 'test_unsupervised';

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

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

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

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

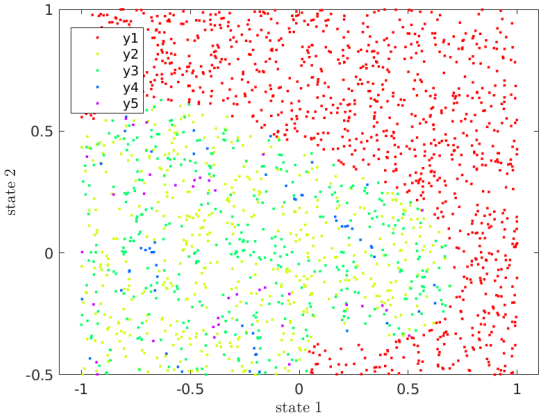
Again, plotting the results is achieved by calling the following routines

```
% 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);
```

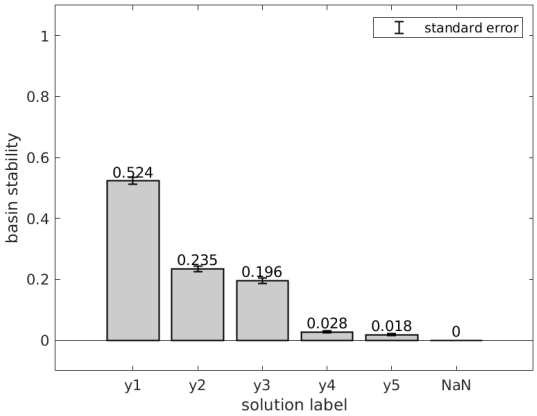
`bSTAB` automatically assigns names `y1`, `y2`, etc. to each of the clusters determined by the DBSCAN method. Plotting the feature space (third plot command above) then reveals how many and which clusters where found, as shown in Figure 5 (a). In this case all clusters are optimally separated, resulting in the correct number of k = 5 multi-stable solutions co-existing in the given system configuration. The state-space in (b) is similar to the one shown in the main publication, but there derived from a-priori knowledge about the number and characteristics of the multi-stable solutions. Particularly relevant is the finally reported basin stability spectrum, shown in Figure 5 (c): the `NaN` class label collects those samples which DBSCAN was not able to assign to one of the clusters, i.e. representing outliers. Sub-optimal clustering will result in a non-zero basin stability value for this class, and hence is an indicator for weak classification performance. In this case, more or other features (i.e. more descriptive and discriminative) may need to be extracted from the steady-state trajectories, or the parameters of DBSCAN require further fine-tuning.



(a)



(b)



(c)

Figure 5: Graphical output after the basin stability computation for the unsupervised Duffing case: (a) feature space resulting from DBSCAN clustering, (b) samples in state space labeled by the respective solution, and (c) resulting basin stability values

References

[1] 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.

[2] THOMPSON, J. M. T., AND STEWART, H. B. *Nonlinear dynamics and chaos*, 2. ed. ed. Wiley, Chichester, 2002.