User Manual of SynNBC

A new software toolbox for the sound and automated synthesis of barrier certifica based on CEGIS and SOS

1 Introduction

SynNBC is a new software toolbox for the sound and automated synthesis of barrier certificates, for the purposes of verifying the safety of continuous-time dynamical systems modelled as differential equations. The tool leverages a counterexample-guided inductive synthesis (CEGIS) engine. Besides the learners and verifiers in CEGIS, we have added SOS post-verification to re-verify the barrier functions with no counterexample having been found.

The directory in which you install SynNBC contains five subdirectories:

- /benchmarks: the source code and some examples;
- /learn: the code of learners;
- /verify: the code of verifiers;
- /plots: the code of plots;
- /utils: the configuration of CEGIS.

2 Configuration

2.1 System requirements

To install and run SynNBC, you need:

- Windows Platform: Python 3.9.12;
- Linux Platform: Python 3.9.12;
- Mac OS X Platform: Python 3.9.12.

2.2 Installation instruction

You need install required software packages listed below and setting up a MOSEK license .

1. Download SynNBC.zip, and unpack it;
2. Install the required software packages for using SynNBC:
   - pip intsall matplotlib==3.5.3;
   - pip intsall numpy==1.23.2;
   - pip intsall scipy==1.9.0;
   - pip intsall SumOfSquares==1.2.1;
   - pip intsall sympy==1.11;
   - pip intsall torch==1.12.1;
   - pip install Mosek==10.0.30;
   - pip install gurobipy==10.0.0
   - pip install picos==2.4.11

3. Obtain a fully featured Trial License if you are from a private or public company, or Academic License if you are a student/professor at a university.
   - Free licenses
     - To obtain a trial license go to `https://www.mosek.com/products/trial/`;
     - To obtain a personal academic license go to `https://www.mosek.com/products/academic-licenses/`;
     - To obtain an institutional academic license go to `https://www.mosek.com/products/academic-licenses/`;
     - If you have a custom license go to `https://www.mosek.com/license/request/custom/` and enter the code you received.
   - Commercial licenses
     - Assuming you purchased a product (`https://www.mosek.com/sales/order/`) you will obtain a license file.

3 Automated Synthesis of Barrier Functions

Main steps to generate verified barrier functions:

1. create a new example and confirm its number;
2. input dimension *n*, three domains: *D_zones*,*I_zones* and *U_zones* and differential equations *f*;
3. define the example's name, call function get_example_by_name, input parameters of *opts* and get verified barrier functions.

3.1 New examples

In SynNBC, if we want to generate a barrier function, at first we need create a new example in the *examples* dictionary in *Exampler_B.py*. Then we should confirm its number. In an example, its number is the key and value is the new example constructed by *Example* function.

```
>> 1:Example()
```

3.2 Inputs for new examples

At first, we should confirm the dimension *n* and three domains: *D_zones*,*I_zones* and *U_zones*. For each domain, the number of the ranges must match the dimension *n* input.

Example 1 Suppose we wish to input the following domains:

$$D\_zones = \{x_1 \geq -2, x_1 \leq 2, x_2 \geq -2, x_2 \leq 2\}$$

$$I\_zones = \{x_1 \geq 0, x_1 \leq 1, x_2 \geq 1, x_2 \leq 2\}$$

$$U\_zones = \{x_1 \geq -2, x_1 \leq -0.5, x_2 \geq -0.75, x_2 \leq 0.75\}$$

This can be instantiated as follows:

```
>>  n=2,
>>  D_zones=[[-2, 2]] * 2,
>>  I_zones=[[0,1],[1,2]],
>>  U_zones=[[-2,-0.5],[-0.75,0.75]],
```

Then, the dynamical system should be confirmed in the *Example* function. The dynamical system is modelled as differential equations *f*. We define the differential equations through lambda expressions. The variables x_1, x_2, x_3, ... , x_n should be typed as x[0], x[1], x[2], ... , x[n-1]. All differential equations are input into the *f* list.

For Example 1, we consider the following differential equations:

$$\begin{cases} \mathrm{f}_1 = x_2+2x_1x_2 \\ \mathrm{f}_2 = -x_1-x_2^2+2x_1^2 \end{cases}$$

Construct the differential equations by setting

```
>>  f = [
            lambda x: x[1]+2*x[0]*x[1],
            lambda x: -x[0]-x[1]**2+2*x[0]**2
        ],
```

3.3 Getting barrier functions

After inputting the dimension, domains and *f*, we should define the example's name. For instance, to create an example named *barr_1*, you need type:

```
>>  name='barr_1'
```

The completed example is following:

```
>>  1:Example(
        n=2,
        D_zones=[[-2, 2]] * 2,
        I_zones=[[0,1],[1,2]],
        U_zones=[[-2,-0.5],[-0.75,0.75]],
        f=[
            lambda x: x[1]+2*x[0]*x[1],
            lambda x: -x[0]-x[1]**2+2*x[0]**2
        ],
        name='barr_1'
    )
```

Then we should update the code of test_barrier.py or create a new python file by imitating its code. For generating a barrier function, we should input the parameter *name* to call function get_example_by_name and set the parameters of *opts*.

For Example 1, the code example is as follows:

```
>> activations = ['SKIP']
>> hidden_neurons = [10] * len(activations)
>> example = get_example_by_name('barr_1')
>> opts = {
        "ACTIVATION": activations,
        "EXAMPLE": example,
        "N_HIDDEN_NEURONS": hidden_neurons,
        "MULTIPLICATOR": True,
        "MULTIPLICATOR_NET": [],
        "MULTIPLICATOR_ACT": [],
        "BATCH_SIZE": 500,
        "LEARNING_RATE": 0.1,
        "LOSS_WEIGHT": (1.0, 1.0, 1.0),
        "MARGIN": 0.5,
        "SPLIT_D": not True,
        "DEG": [2, 2, 2, 1],
        "R_b": 0.6,
        "LEARNING_LOOPS": 100,
        "CHOICE": [0, 0, 0]
   }
```

The options of *activations* include "SQUARE","SKIP" and "MUL". "MULTIPLICATOR" of *opts* means whether to use multipliers. "LOSS_WEIGHT" represents the weights of init loss, unsafe loss, and diffB loss. "SPLIT_D" indicates whether to divide the region into 2^n small regions when searching for counterexamples, with each small region searching for counterexamples separately. "CHOICE" indicates a choice between scipy.optimize.minimize function and gurobi solver. (0 represents using minimize function and 1 represents using gurobi solver)

At last, run the program and we can get verified barrier functions. For Example 1, the result is as follows:

```
B = 0.364960715038238*x1**2 - 0.251598987567664*x1*x2 -

    0.750793246501246*x1 - 0.661513285579136*x2**2 -

    0.848561513601891*x2 + 0.561290025252521
```