IraFhy

User Manual

Jianqiang Ding, Xue Bai

September 28, 2019

Contents

1 Introduction

In this section, we will briefly introduce the background of the project, especially about current most popular tools in the field of reach-ability analysis. It will be illustrated what inspired us to start the project, and what the project designed for. The structure of the documentation can be found in the subsection 1.3.

1.1 Related Works

Reach-ability analysis, which involves constructing reachable sets, is a central component of model checking, plays an important role in automatic verification and falsification of safety properties for continuous nonlinear and hybrid systems. Over the past years, lots of tools like CORA, HyPro, FLOW* were developed as formal verification tools for hybrid systems. The main differences among these tools concentrated on the following aspects:

state set representation because the intersection and merging operations between state sets are involved in the calculation process, appropriate representation of state sets will not only affect the calculation efficiency of the algorithm, but also affect the accuracy. For various reasons, different tools will determine the appropriate state set representation according to specific requirements.

IVP solver different approaches to determine the set of system states that are reachable from a given set of initial states will also affect the efficiency and speed of the algorithm. Especially, for a specific algorithm, it is necessary to set a specific module to address the initial value problem.

verification method when discussing the reach-ability of hybrid system, although the duration of the analysis is specified, it is necessary to verify the intermediate results of the running process, e.g. when discussing the safety of hybrid system, it is necessary to verify whether the running state set intersects with the unsafe set, and then the running of the algorithm is generally terminated.

1.2 Motivation

Most of current tools focus on over-approximation or under-approximation computing, Nevertheless, most of them only focus on the implementation of some specific algorithms, if someone want to test a completely new idea about hybrid systems' reach-ability analysis, one may need to write lots of code from bottom to top, or assemble lots of old or new tools as modules which hardly guarantee the compatibility with each other, into one program. All of these "dirty works" really will killing and time wasting. The idea of the framework is providing such a framework which allows users can customize and test their algorithms about reach-ability analysis easily, fewer code, more clear computing procedure, so that allowing researchers pay more attention to refine the core of their research.
To cut a long story short, what we want to do is, providing a framework which can present a paper's idea clearly to make it easy to be captured by others while the authors only need to write relatively small amount of core code.

1.3 Overview

This documentation is structured as follows, background and motivation of the project are introduced in section 1, dependencies and detailed configurations about compilation and installation are provided in section **??**, customizing one's own algorithm, modifying related test cases, and setting of results viewing are illustrated in section 3. State set representations using inside the project are detailed in section 4. All the utilities like parser, solver and optimizer and so on, can be found at section 5.

2 Getting Started

This section will show you the dependencies of the library, and how to modify the compilation and installation.

2.1 Dependencies

IraFhy depends on the following third-party libraries:

Third-party libraries		
Library	URL	Version
Eigen	http://eigen.tuxfamily.org/index.php?title=Main_Page	3.34
CMake	https://cmake.org/	latest
GLPK	https://www.gnu.org/software/glpk/	4.65-2
FILIB++	http://www2.math.uni-wuppertal.de/wrswt/software/filib.html	3.0.2
boost	https://www.boost.org/	1.67.0.0
Doxygen	http://www.doxygen.nl/	1.8.13
Graphviz	https://www.graphviz.org/	2.40.1-5
Google Test	https://github.com/google/googletest	1.8.1
Qhull	https://github.com/qhull/qhull	7.3.2
FLANN	https://www.cs.ubc.ca/research/flann/	1.9.1
Antlr	https://www.antlr.org/	4.7.2
libigl	https://github.com/libigl/libigl	2.1.0
OpenMP	https://www.openmp.org/	latest

🎓 All of the tests and examples run on Ubuntu 19.04, other Linux distributions should have no difference.

2.2 Installation

It is recommended build the library ***out of source***, e.g. build in a separate folder named ***build***. We will under the assumption that the folder ***build*** has been created under the root directory to illustrate the configuration options and installation.
So we recommend you run the following command under the root directory of the source code before compilation.

```
$ mkdir build && cd build
```

2.2.1 Configuration

The IraFhy provides such following configuration options:

- **ENABLE_EXAMPLES** enable generating examples
- **ENABLE_TESTING** enable Unit Testing

- **ENABLE_DOC** enable generating documents

2.2.2 Building

You may specify the building procedure with different options as following: e.g. under the ***build*** directory, if someone want to enable generating examples, unit testing and documents generating, simply runs the following command:

```
$ cmake .. –DENABLE_EXAMPLES=1 –DENABLE_TESTING=1 –DENABLE_DOC=1
```

Once the cmake configuration and generation procedure done, the *example_0* can be compiled and runs as following:

```
$ make example_0
```

Documents can be generated by running the command:

```
$ make doc
```

Unit testing can be turned on by running the following command:

```
$ make test
```

3 Tutorial

In this section, we will introduce you how to use this library to define your own algorithm and how to write the test cases, at the end of this section, some options about the results plotting were provided.

3.1 customize your own algorithm

For convenience, the concept of hybrid systems is referred to in the article [1]. Almost all separable modules in the accessibility process are decoupled into separate computing modules. However, at present, we only allow users to build specific accessibility analysis algorithms by customizing three modules as follows:

3.1.1 Analyser

The analyzer module is mainly used to solve the initial value problem, that is to say, given the specified initial state set and the specified duration, as well as the given continuous dynamic system, the Analyser module should return the state set that the continuous dynamic system can reach. Different reach-ability analysis algorithms usually define their own specific calculation methods to get the reachable set in a certain time interval. Therefore, we public the module, so that users can customize their own reach-ability set algorithm by customizing the actual calculation method within the Analyzer module.

e.g. if you just want to get the over approximate state set in a duration, you can simply call the solver which can return the over approximate set as following:

```
#pragma once

#include <irafhy/analyser.h>
#include <irafhy/representation/geometric/intervalHull.h>
#include "../settings/settings.h"

namespace irafhy
{
  class OAFAnalyser : public irafhy::Analyser
  {
  public:
    capd::C0Rect2Set compute(const irafhy::Time&    duration,
                 const irafhy::System&   system,
                 const capd::C0Rect2Set& initCondition,
                 const Settings&      settings) const override;
  };
} // namespace irafhy
```

source code of ./algorithm/overApproximateForward/analyser/analyser.h

```
#include "analyser.h"
#include <irafhy/utility/solver/ODESolver.h>

```

```
namespace irafhy
{
  capd::C0Rect2Set OAFAnalyser::compute(const irafhy::Time&
    duration,
                           const irafhy::System&   system,
                           const capd::C0Rect2Set& initCondition,
                           const irafhy::Settings& settings) const
  {
    try
    {
      return ODESolver::solve(system, initCondition, duration);
    }
    catch (const std::bad_cast& e)
    {
      std::cout << e.what() << std::endl;
      exit(EXIT_FAILURE);
    }
  }
} // namespace irafhy
```

source code of ./algorithm/overApproximateForward/analyser/analyser.cpp

As you can see, there is nothing special within the analyser but an ODE solver calling. All you need to do is to define your own class which inherited from the pre-defined ***Analyser*** and overwrite the pure virtual method ***compute***. You can also define a relatively complex analyser class like this which implement the algorithm in article [2]

```
#pragma once

#include <irafhy/analyser.h>
#include <irafhy/representation/geometric/intervalHull.h>
#include <irafhy/representation/geometric/polytope.h>
#include "../settings/settings.h"
#include <omp.h>

namespace irafhy
{
  class UABPAnalyser : public Analyser
  {
  private:
    [[nodiscard]] std::vector<IntervalHull> boundary(const
    Condition& condition, double epsilon) const;
    [[nodiscard]] std::vector<IntervalHull> simulate(const Time&
                duration,
                                const System&               system,
                                const std::vector<IntervalHull>& boundaryIntervalHulls) const;
    [[nodiscard]] Polytope          constructPolytope(const std::vector<IntervalHull>& boundary) const;
    [[nodiscard]] Polytope          contraction(const Polytope& polytope,
                                 const std::vector<IntervalHull>& boundary,
                                 bool                isExact) const;

  public:
    [[nodiscard]] capd::C0Rect2Set compute(const Time& duration,
                            const System&        system,
                            const capd::C0Rect2Set& initCondition,
                            const Settings&        settings) const override;
  };
} // namespace irafhy
```

source code of ./algorithm/underApproximateBackwardUsingPolytope/analyser/analyser.h

3.1.2 Verifier

The verifier module is another module you need to customize before testing your own algorithm. This module responsible for the verification of the running procedure. It is used to control whether the procedure should be terminated. The verification method will be called after each step. Resulting **TRUE** means the algorithm can be run one more step. **FALSE** means the algorithm should not be running any more. You can customize your own verifier just like what we do in subsection 3.1.1.

```
#pragma once

#include <irafhy/verifier.h>
#include <irafhy/representation/formal/hybridAutomaton/hybridautomaton.h>

namespace irafhy
{
  class OAFVerifier : public Verifier
```

```cpp
  {
  public:
    bool verify(const Settings& settings, const void* hybridAutomaton) const;
  };
} // namespace irafhy
```

source code of ./algorithm/overApproximateForward/verifier/verifier.h

```cpp
#include "verifier.h"

namespace irafhy
{
  bool OAFVerifier::verify(const Settings& settings, const void* hybridAutomaton) const { return true; }
} // namespace irafhy
```

source code of ./algorithm/overApproximateForward/verifier/verifier.cpp

for more complex verifier example, you can check the source code of the repository.

3.1.3 Settings

It is inevitable to set some parameters related to the algorithm. For this part, we use **Settings** class to maintain all of them. You can customize your own **Settings** class to maintain the parameters you need and add some other methods you want. e.g.

```cpp
#pragma once

#include <irafhy/settings.h>

namespace irafhy
{
  class OAFSettings : public Settings
  {
  public:
    OAFSettings(const Settings& settings);

    void help() const override;
  };
} // namespace irafhy
```

source code of ./algorithm/overApproximateForward/settings/settings.h

⚠ It should be noted that the **Analyser**, **Verifier**, **Settings** should be in the namespace **irafhy**.

3.2 configure the test case

All the configurations related to the test case should be defined into two files, one for ***Hybrid System***, the other for ***Settings***. e.g.

```
HYBRID_AUTOMATON
{
    NAME Fitz Hugh Nagumo neuron model

    VARIABLES
    {
        x y
    }

    LOCATIONS
    {
        MODEL
        {
            NAME electron
            FLOWS
            {
                x'=x-x^3-y+(7/8)
                y'=0.08*(x+0.7-0.8*y)
            }
            INVARIANT_CONDITIONS
            {
            }
        }
    }

    TRANSITIONS
    {
    }
}
```

model.mdl

for the content of the **Settings**, you should declare all the parameters you need inside the configuration file like this.

```
SETTINGS
{
    TIME_HORIZON := [0,0.2]
    STEP := 0.02
    GEOMETRY := INTERVAL_HULL
    PLOT := ON
    PRINT := OFF
    ANALYSIS := FORWARD
    START_MODEL_ID := electron #name of the model
    INITIAL_CONDITION :=
    CONDITION
    {
        INTERVAL_HULL
        {
            [0.5,1.5],
            [2.0,3.0]
        }
    }
}
```

setting.cfg

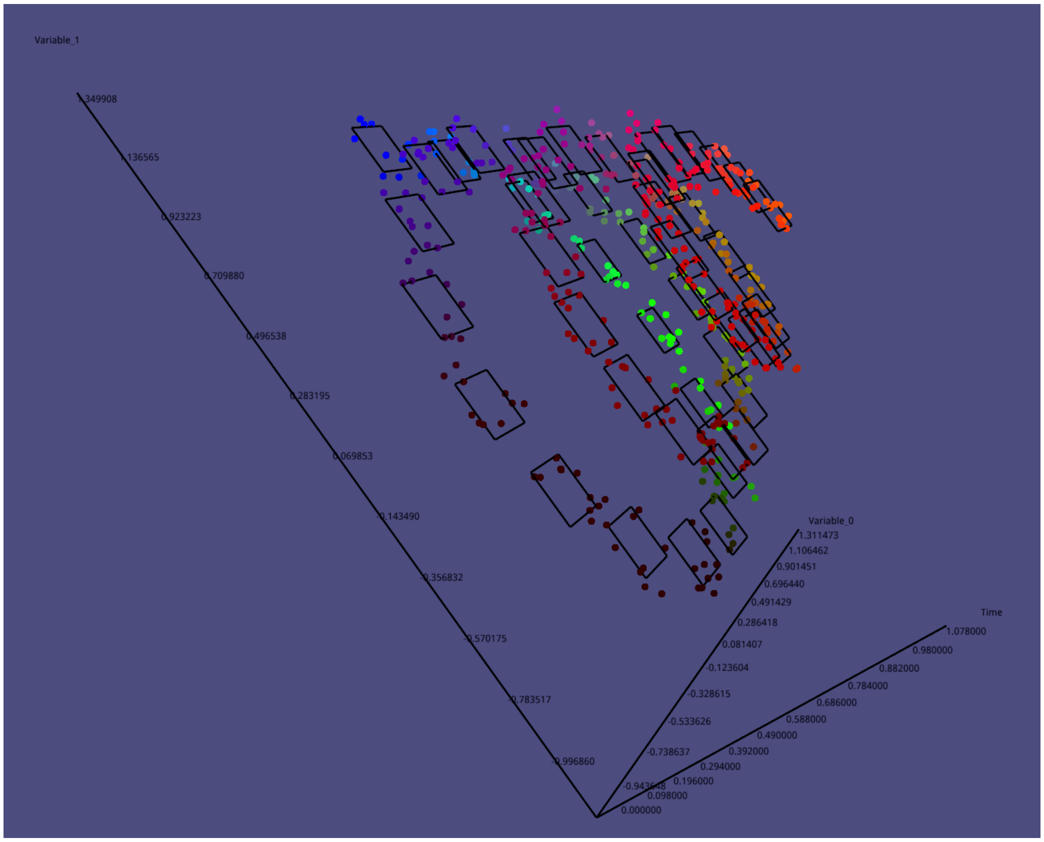
once two files done, the parameter inside shall be parsed by the parser only if you use the right name of the parameter and the set it to the right value.

3.3 view the results

The results can be plotted once the results generated, you can simply call the **show** method of class **viewer** to view them. The viewer current is generally an unpolished tool which supports limited viewing options, it will be improved future.

```
irafhy::viewer::show(
          this->intervalHulls_,
          this->points_,
          this->time_, {0, 1, 2},
          irafhy::VIEW_TYPE::LINE);
```

As you can see, you should provide the geometry objects, points, the time intervals, the dimensions and the style of the plotting. here is a screenshot showing a bunch of 3D interval hulls, some points and time intervals which indicate when these geometry objects generated. Even the **dimension** values set to **{0,1,2}**, the viewer still just show 2D interval hulls cause the time intervals provided.



4 State set representations:

This section will detail the formal representations and the geometry representations used inside our library. We only support interval hulls and ploytope so far. More geometry objects shall be supported in the future.

4.1 Formal representations

4.1.1 Condition

Condition is designed as the general formal representation of the geometry objects used in computing. The entity of the condition can be interval hull, polytope or any other geometry objects shall be supported in the future.

4.1.2 Constraint

Constraint is designed as the general formal representation of the set of state. It is mainly used in declaring a geometric state set formally, e.g. you can define the polytope using formulations as following:

$$\begin{cases} A_0 \cdot x & + & b_0 & \leq 0 \\ A_1 \cdot x & + & b_1 & \leq 0 \\ \vdots & & & \\ A_n \cdot x & + & b_n & \leq 0 \end{cases}$$

which defines a convex hull as a polytope (*half spaces defined*).

4.1.3 System

System is designed to support describing continuous dynamic system formally. The ***system*** responsible for receiving the formal description of a system outside and generating the functions can be solved by the ODE solver inside.
It is pretty simple to define a continuous dynamic system, all you need to do is writing down all the functions into the configuration file like 3.2 do, the configuration will be parsed, and the system will be constructed by the parser automatically.

4.1.4 Hybrid Automaton

Hybrid Automaton[1] is implemented as deterministic finite automaton[3] inside. A hybrid automaton is a tuple

$$\boldsymbol{H}=(\boldsymbol{Locations},\boldsymbol{Variables},\boldsymbol{Flow},\boldsymbol{Invariants},\boldsymbol{Transitions},\boldsymbol{Initialization})$$

Locations a finite set of discrete states which are also called control modes

Variables a finite ordered set of real-valued variables

Flow continuous dynamics associate with each mode defined by ordinary differential equations

Invariants state set which associates assigned to each mode

Transitions finite set of discrete transitions which specify jumps among modes

Initialization initial state set of the starting mode of the automaton

🎓 For more details you can check the paper[1]

4.2 Geometric representations:

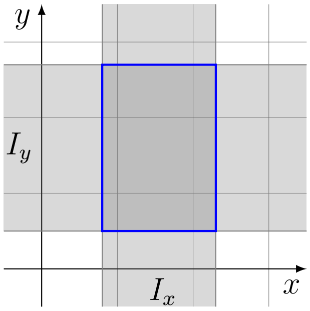
This section details the implementation of the data structures for representation of state sets and related operations. Variety designing of representations is mainly to balance computational complexity and accuracy. While some representations are able to perform certain operations very efficiently, other operations on the same representation, which are also needed for the analysis, can be computationally expensive. We only support ***interval hull*** and ***polytope*** by now. All the geometry representations inherit from the abstract ***Geometry*** class, all the pure virtual **API** should be overrided if you wanna define your own geometry representation. here are the **API**s:

```
#ifndef REPRESENTATION_GEOMETRIC_GEOMETRY_H
#define REPRESENTATION_GEOMETRIC_GEOMETRY_H

#include <Eigen/Core>

namespace irafhy
{
  class Point;

  template <typename DerivedGeometry>
  class Geometry
  {
  public:
      virtual ~Geometry()                              =
    default;
      virtual int         dimension() const                  =
     0;
      virtual bool        empty() const                    =
    0;
      virtual bool        intersect(const DerivedGeometry&,
    DerivedGeometry& result) const = 0;
      virtual DerivedGeometry unite(const DerivedGeometry&) const
                  = 0;
      virtual bool        contains(const Point& point) const
              = 0;
      virtual bool        contains(const Eigen::VectorXd& coordinate
    ) const          = 0;
  };
} // namespace irafhy
#endif //REPRESENTATION_GEOMETRIC_GEOMETRY_H
```

source code of ./include/irafhy/representation/geometric/geometry.h

4.2.1 Intervalhull

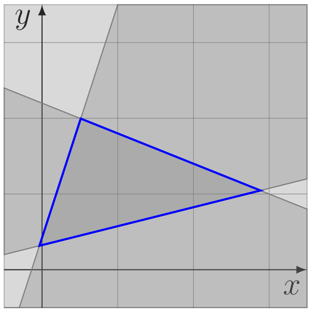
An ***interval hull*** is represented by a group of intervals, one for each dimension of the state space.



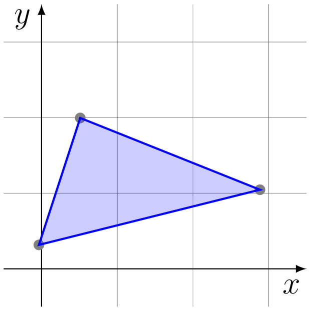
4.2.2 Polytope:

We restrict ourselves to closed convex polytope, and it can be divided into the following two categories according to the different representations.

HPolytope defined by a set of half spaces, which given by a normal matrix A and related offset vector B



VPolytope defined by a set pf points which specify the convex hull of those points.



5 Utilities

This section provides the brief introductions of all auxiliary functions used in the framework, help you to grasp the structure of the framework better.

5.1 Computing Geometry

5.1.1 Convex hull

We wrap the third party library ***Qhull*** to support convex hull related functions, more details about ***Qhull***, please check its official website.

5.2 Extension

The extension maintains all the auxiliary functions used to extend the standard third party libraries. It consists of extensions to **Eigen** and **Qhull** so far, to support wrapping the third party libraries easier.

5.3 Optimizer

It is inevitable to doing optimization during computation, the framework organizes these third-party optimizers under the directory *optimizer*. We only support linear optimization so far by wrapping **GLPK**, which is a package is intended for solving large-scale linear programming and other related problems.

5.3.1 Linear Programming

It is easy to define a linear programming problem which can be solved by the wrapped **GLPK**, e.g.

$$maxmize\ z = 10 \cdot x_1 + 16 \cdot x_2 + x_3 + 0.0$$

$$subject\ to\ \begin{cases} 1 \cdot x_1 & +1 \cdot x_2 & +1 \cdot x_3 \leq 100 \\ 10 \cdot x_1 & +4 \cdot x_2 & +5 \cdot x_3 \leq 600 \\ 2 \cdot x_1 & +2 \cdot x_2 & +6 \cdot x_3 \leq 300 \end{cases}$$

$$where\ all\ variables\ are\ non-negative$$

$$x_1 \geq 0, x_2 \geq 0\ x_3 \geq 0$$

The problem can be defined by constraint matrix and vector, once the solving done, you can check the *Evaluation* instance returned to get the solution. you can check the problem related test code under the directory *test*.

5.4 Parser

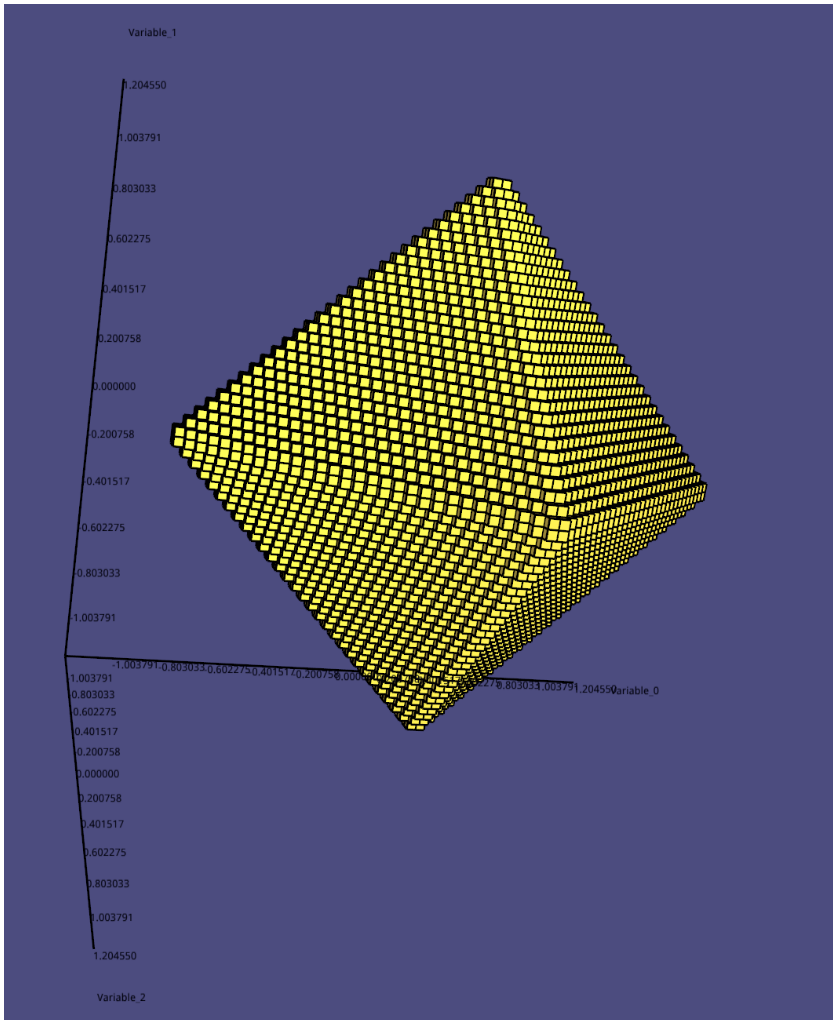
The parsers are defined with the help of **Antlr**, which is a powerful parser generator. for more details about **Antlr**, please check the official document.

5.5 Solver

There are so many solvers written in different languages, rigorous or non-rigorous. We restrict to rigorous ordinary differential equations solver and constraints sanctification problem solver.

5.5.1 Constraints Sanctification Problem solver

IBEX is the most current active library designed for constraint processing over real numbers in C++, previously, **Realpaver** is one of the most popular library for nonlinear constraint solving and rigorous global optimization(it seems that without maintenance any more). most of these libraries lack of providing **API**s for getting boundary interval hulls of the target domain. So we implement our own solver which can get the boundary interval hulls of the target domain.
e.g. Given the constraints of an octahedron, you can get all the boundary interval hulls as showed in the screenshot following:



5.5.2 Ordinary Differential Equations solver

We wrap the third party library CAPD as our ODE solver. for more details about CAPD, please check its official website.

5.5.3 Viewer

We provide a rough viewer to help the users to check the results generated during the computation, by wrapping the viewer of library *libigl*[4] for more details, please check its official website.

5.5.4 Printer

THE MODULE STILL UNDER DEVELOPMENT.

5.5.5 Plotter

THE MODULE STILL UNDER DEVELOPMENT.

References

[1] Thomas A Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[2] Bai Xue, Zhikun She, and Arvind Easwaran. Under-approximating backward reachable sets by polytopes. In *International Conference on Computer Aided Verification*, pages 457–476. Springer, 2016.

[3] Wikipedia contributors. Deterministic finite automaton — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Deterministic_finite_automaton&oldid=915398161`, 2019. [Online; accessed 15-September-2019].

[4] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/.