SCoRe

Release 1.0.1

Julian Brew

Nov 04, 2019
This is the SCoRe Python library documentation. This software is able to compute samples of reachability sets and tubes over both forward and backward time using continuation and homotopy methods.
1.1 Introduction

This tutorial walks through the process of performing a reachability analysis on a dynamic system.

A typical reachability analysis could involve computing forwards/backwards reachable sets/tubes for a given dynamic system under control and initial condition constraints.

In general, the steps to perform a reachability analysis using this toolbox are

1. Import necessary modules, functions, variables, etc.
2. Create dynamics object
3. Create initial condition constraint object
4. Create reachability object and initialize particles
5. Compute reachability
6. Compute trajectories
7. Plot results
8. Further Analysis
9. Save data

1.2 Example Scenario

In this tutorial, we will look at the classic double integrator dynamic system given by

\[
\ddot{x} = u, \quad \dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u
\]  (1.1)

With control input constraints given by

\[|u| \leq 1\]  (1.2)

and initial condition constraint given by

\[V(x_0, t_0) = \begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix}^T \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix} - 1 \leq 0\]  (1.3)
1.2.1 Imports

To create a reachability object requires a dynamics object and an initial condition constraint object. Let’s start by importing the Constraint object, dynamic system object, Reachability object, and plotting functions:

```python
import numpy as np
from dynSystems import LTIDynamics
from constraints import Constraint
from reach import Reachability
from plotting import plotReachTrajectories, plotReachVolume, plotReachHistories1D,
    plotReachGrowth
```

1.2.2 Create Dynamic System Object

In this case, the dynamic system is linear time-invariant (LTI). A `LTIDynamics` object constructor is provided. All we must provide are the constant A, B matrices as numpy arrays and the limitations on the control input. Because the control input is a scalar in this example, the control input constraint may be specified by providing a scalar value (or list with single scalar entry) in the `uLimit` argument:

```python
A = np.array([[0.0, 1.0], [0.0, 0.0]])
B = np.array([[0.0], [1.0]])
LTI = LTIDynamics(A, B, uLimit=1.0)
```

This dynamic system object is the interface for the reachability object. The dynamic system object should contain methods for computing state dynamics, costate dynamics, optimal control, and also for propagating dynamics given initial conditions.

1.2.3 Create Initial Condition Constraint

The initial conditions are constrained by an ellipsoidal-type constraint. The `Constraint` class has a constructor for ellipsoidal constraints where the center and shape matrix are specified:

```python
E = np.diag([0.5, 0.5])
IC = Constraint.fromShapeMatrix(E, p = 2)
```

This constraint object is used by the reachability object (among others) to evaluate and compute properties related to points that lie on the boundary of or within the specified constraint.

1.2.4 Create Reachability Object and Initialize Particles

Now that a initial condition constraint and dynamic system have been defined, we can now create a reachability object:

```python
R = Reachability(LTI, IC)
```

The `Reachability` object represents a single reachability analysis. This object interfaces with both the dynamics object for particle flow propagation and initial condition constraint for evaluating constraint functions.

One can now populate the initial condition constraint with a specified number of particles by the following. We can also specify the number of dimensions for the state space subspace that we are going to perform the reachability analysis over:

```python
R.initializeReach(subspaceDim=2, numParticlesPerDim=50)
```
This creates 50 particle objects that sample the boundary of the initial condition constraint. These particles also contain graph information of its nearest neighbors.

### 1.2.5 Compute Reachability over Time Horizon

Now we are ready to use the particles and reachability object to compute how these reachable set samples evolve with respect to time horizon. In order to do so, we must create an array of time horizons to compute the flow of particles over. The time horizon array must be monotonically increasing or decreasing. If the time horizon array is increasing, a forward reachability analysis is performed and vice versa for a backwards reachability analysis:

```python
timeSteps = 30
t0 = 0.0
tf = 2.0
Tvec = np.linspace(t0, tf, timeSteps)
R.computeReach(Tvec)
```

This uses numerical continuation methods to evolve the optimal trajectory initial conditions over time horizon for each particle.

### 1.2.6 Compute Trajectories

Once the optimal initial conditions are computed, a simple propagation of particle trajectories is required to plot and perform analysis on the reachable set. This can be performed using the following lines of code:

```python
R.computeCurrentStateTrajectories(numTimeSteps=timeSteps)
R.computeFinalStateHistories()
```

### 1.2.7 Plot results

There are a large number of visualizing 2D and 3D reachability results provided. Examples of a few are shown here:

```python
fh1 = plotReachTrajectories(R.particles)
fh2 = plotReach(R.particles)
fh3 = plotReachGrowth(R.particles)
fh4 = plotReachTrajectories1D(R.particles, projDim=1, shadedTrajBool=True, 
    particleTrajBool=True)
fh5 = plotReachTrajectories1D(R.particles, projDim=2, shadedTrajBool=True, 
    particleTrajBool=True)
```
1.2.8 Further Analysis

Once a reachable set is computed, there are a number of operations that can be performed for further analysis. For example, one can convert the computed reach set to a reach tube using the following:

```python
R2 = R.convertReachSetToTube(returnNew=True)
fh6 = plotReachGrowth(R2.particles)
```

If desired, one may also add additional particles for more uniform around low curvature regions of the reachable set boundary:

```python
R.meshRefinement(method='bisect', q1=25, q3=55, maxRefinements=4)
fh7 = plotReachTrajectories(R.particles, addedPtInds=list(range(R.origNumParticles, R.numParticles)))
```

1.2.9 Save Data

If you want to save the figures resulting from your reachability analysis, you can do so using:

```python
```
Fig. 2: Forward reachable tube converted from reachable set

Fig. 3: Additional particles added through bisection mesh refinement
The easiest way to save the entire reachability analysis is to use the dill module:

```python
import dill
saveFilename = 'doubleIntegrator_session.pkl'
dill.dump_session(saveFilename)
```

This saves the entire python workspace for later use. To load the saved file:

```python
import dill
dill.load_session('doubleIntegrator_session.pkl')
```
Commonly used notation or nomenclature is defined below. There are a few instances where certain terms are used in multiple ways. The hope is that context should be enough to distinguish the meaning of the term.

### 2.1 Reachability terms

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachable Set or RS</td>
<td>Volume of state space representing all achievable states exactly at the given time horizon T originating from the initial condition volume.</td>
</tr>
<tr>
<td>Reachable Tube or RT</td>
<td>Volume of state space representing all achievable states up to the given time horizon T originating from the initial condition volume.</td>
</tr>
<tr>
<td>Forward Reachable Volume</td>
<td>Reachable volume of state space where $t_0 &lt; t_f$. In this case, the reachable volume must originate from the initial condition set and change as $t$ goes from $t_0$ to $t_f$.</td>
</tr>
<tr>
<td>Backwards Reachable Volume</td>
<td>Reachable volume of state space where $t_f &lt; t_0$. In this case, the reachable volume must terminate at the initial condition set and change as $t$ goes from $t_f$ to $t_0$.</td>
</tr>
<tr>
<td>Subspace Reachable Volume</td>
<td>Volume of subspace of state space representing all achievable subspace states at or up to the given time horizon T originating from the initial condition volume. This is equivalent to projection of the full state space reachable volume.</td>
</tr>
<tr>
<td>Subspace of interest</td>
<td>Subspace of state space that one wants to compute reachable volumes in. This can also be equivalent to the full state space if desired. PLEASE NOTE, the subspace has to be from the first few components of the state for this software. In other words, if you want to compute reachability with respect to a particular component of the state vector, you should rearrange the state, dynamics, initial condition constraint, etc. so the important state components/subspace is listed first in the state vector.</td>
</tr>
<tr>
<td>subspaceDim</td>
<td>Number of dimensions in the subspace of state space to perform the reachability analysis</td>
</tr>
<tr>
<td>Performance Metric or V</td>
<td>The objective function in the optimal control problem definition. This is usually defined using inner products with the final state and a unit vector search direction</td>
</tr>
<tr>
<td>Particle or Sample</td>
<td>A sample of the reachable volume boundary. Each particle corresponds to the solution to a single optimal control problem (parametrized by a unit vector search direction)</td>
</tr>
</tbody>
</table>
### 2.2 Vector terms

Table 2: Vector Notation Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N or n</td>
<td>State space dimension</td>
</tr>
<tr>
<td>M or m</td>
<td>Control input dimension</td>
</tr>
<tr>
<td>x or x</td>
<td>State space vector</td>
</tr>
<tr>
<td>p or p</td>
<td>Costate vector</td>
</tr>
<tr>
<td>y or y</td>
<td>Trajectory flow state vector, y = [x, p]</td>
</tr>
<tr>
<td>Phi</td>
<td>Trajectory flow state transition matrix (STM), STM for y</td>
</tr>
<tr>
<td>Y or Y</td>
<td>Augmented trajectory flow state vector, Y = [x, p, vec(Phi)]</td>
</tr>
<tr>
<td>lam or l</td>
<td>Lagrange multiplier for initial condition constraint</td>
</tr>
<tr>
<td>z</td>
<td>Optimal initial solution, z = [x0, lam]</td>
</tr>
<tr>
<td>ds</td>
<td>Unit vector search direction in optimal control problem definition</td>
</tr>
<tr>
<td>th</td>
<td>Hyperspherical coordinate representation of ds</td>
</tr>
</tbody>
</table>

### 2.3 Time notation

Table 3: Time Notation Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Time horizon for system to evolve</td>
</tr>
<tr>
<td>t</td>
<td>Time where t goes between t0 and tf</td>
</tr>
<tr>
<td>Initial time</td>
<td>Time where the state space constraint volume is known/specified. This is the time where the possible/feasible states are known/specified</td>
</tr>
<tr>
<td>Final/Terminal time</td>
<td>Desired time when reachable volume should be computed.</td>
</tr>
<tr>
<td>_f</td>
<td>Quantity at end of time integration/propagation and at final time, e.g. xf = x(tf)</td>
</tr>
<tr>
<td>_0</td>
<td>Quantity at beginning of time integration/propagation and at initial time, e.g. x0 = x(t0)</td>
</tr>
<tr>
<td>__T</td>
<td>Quantity at/given specified time horizon T, e.g. x0_T = x0(T) = x(t0;T)</td>
</tr>
<tr>
<td>Trajectory</td>
<td>How a quantity changes over time t</td>
</tr>
<tr>
<td>History</td>
<td>How a quantity changes over time horizon T</td>
</tr>
</tbody>
</table>
Optimal control review is listed here

For a majority of this software, the reachability optimal control problem is defined as

$$\max_{u \in U} V(x_f, t_f) \text{ s.t. } \dot{x} = f(x, u, t)g(x_0, t_0, x_f, t_f) = 0$$  \hspace{1cm} (3.1)

where $V(x_f, t_f)$ denotes the performance metric or objective function for the optimal control problem, $U$ denotes the control input constraint, and $g(x_0, t_0)$ denotes the initial condition constraint.

The optimal control input can be computed using Pontryagin’s maximum principle as

$$u^* = \arg\max_{u \in U} p^T f(x, u, t)$$  \hspace{1cm} (3.2)

where $p$ denotes the optimal control costate/adjoint vector.

In general, there is not a closed form solution to the above problem. However, many problems and dynamic systems in engineering have the following form with an analytic solution for the optimal control.

### 3.1 Control Affine Dynamics

Given a continuous-time control affine nonlinear dynamic system of the form

$$\dot{x} = f_1(x, t) + f_2(x, t)u$$  \hspace{1cm} (3.3)

In the case where the control input constraint is of the scaled $p$-norm type such that

$$u = M^{-1}\tilde{u}\|\tilde{u}\|_p \leq 1$$  \hspace{1cm} (3.4)

as described in the Constraint Class. The analytic solution for the optimal control for values of $p > 1$ is then given by

$$c = M^{-T}f_2^T(x, t)p^s = \sum_i c_i \frac{|c_i|^p}{s^p} \tilde{u}_i^* = \frac{\text{sign}(c_i)|c_i|^\frac{1}{p}}{s^\frac{1}{p}} u^* = M^{-1}\tilde{u}^*$$  \hspace{1cm} (3.5)

For the special case of $p = 2$,

$$\tilde{u}^* = \frac{c}{\|c\|_2} u^* = M^{-1}\tilde{u}^*$$  \hspace{1cm} (3.6)

As $p \to \infty$ the optimal control solution approaches

$$\tilde{u}^* = \text{sign}(c)u^* = M^{-1}\tilde{u}^*$$  \hspace{1cm} (3.7)
REACHABILITY CLASS

The Reachability class contains most of the typical functions and variables required for a reachability analysis.

To create a reachability object you need to define an initial condition constraint object and a dynamics object. Refer to these pages of the documentation for typical functions within these classes.

The primary attributes inside the reachability class related to particles/samples are:

- `self.particles` - List of particle objects. Each index of this particle list corresponds to the ID number of the particle.

- `self.neighborsLedger` - List of unique edges in mesh/graph/network. Every index of this list contains the list `[pi_ID, pj_ID]` where `pi_ID` and `pj_ID` correspond to the particle ID numbers (index of `particles` list). This is useful when performing iterations over every pair of particles.

- `self.neighborsList` - List of neighborhoods per particle. Every `i`th index of this list contains the list `([pi_Neighbor_ID1, pi_Neighbor_ID2, pi_Neighbor_ID3, ...])` of particle ID numbers of the neighbors of particle `i`.

- `self.numParticles` - Current number of particles in the mesh/graph for this reachability analysis.

The other important attributes inside the reachability class:

- `self.squaredCost` - Boolean on whether or not to use the squared inner product performance metric as optimal control objective function or the inner product

- `self.subspaceDim` - This specifies the dimension of the subspace of interest in the reachability analysis. This essentially defines the dimension that the particles are sampled in.
The particle class contains functions and attributes to particles in the reachability analysis. These particles should be samples to the reachable volume at particular time horizons.

Every particle is defined by the unit vector search direction $d_s$ that defines which direction in state space the particle should optimize reachability over. Another quantity that is closely tied to this search direction is the angular coordinate vector defined using hyperspherical coordinates that also represent the search direction.

The particles attempt to sample the subspace reachable volume. The particles still operate over the full dimensional state space, but the optimal control objective/performance metric is defined over the subspace of state space that is specified (by `Reachability.subspaceDim`).

If the subspace of interest is $m$-dimensional, then the first $m$ states in the state space vector should be the subspace of interest. By construction of the reachability algorithm, the first reachability analysis is always performed on the first elements of the state space vector.

For the particle class, there is a difference between time horizon $T$ and time $t$. For a specified time horizon, $T$, it is possible to calculate a trajectory (e.g. $x(t)$) from $t = t_0$ to $t = T$. This is because reachability volumes are functions of time horizon while the individual particle trajectories are functions of time $t$.

Each particle object implicitly have an integer ID based on it’s index in `Reachability.particles` list. Particle objects also have neighbors. Neighbors are initially defined by distribution on the unit hypersphere. As particles are spawned and updated, neighbors are defined using Euclidean distance from other particles.

The primary attributes inside the particle class are:

- $d_s, th$ - $d_s$ is the unit vector search direction for optimal control performance metric and support function. $th$ is the hyperspherical coordinate representation of $d_s$
- $x, p, y$ - State $x$, costate $p$, and flow state $y'=[x', p]$ at the current time horizon, $T$
- $z$ - Optimal initial solution vector $z = [x0, lam]$ where $x0$ is optimal initial state (at initial time, $t0$) that satisfies the initial condition constraint and $lam$ is the corresponding lagrange multiplier to the initial condition constraint
- $dFdz$ - Jacobian of necessary condition of optimality function $F$ with respect to $z$
- $dFdT$ - Jacobian of necessary condition of optimality function $F$ with respect to time horizon $T$
- $x_f_T, p_f_T, z_T$ - Final state and costate (at end of time interval) and optimal initial solution as a function of time horizon. $numTimeHorizonSteps$ corresponds to the number of time horizon $T$ values
- $x_t, p_t$ - State and costate trajectories from initial time $t0$ to the current time horizon $T$. $numTimeSteps$ corresponds to the number of time interval values $t$ at current time horizon $T$
- $y_{t,T}$ - Array that stores all the state/costate trajectories over time interval $(t)$ given different time horizon values, $T$
This class represents p-norm constraints and provides functions for evaluating useful quantities related to the constraint. These types of constraints occur often in defining both initial condition and control input constraint regions. In theory, as $p$ approaches infinity, the maximum norm is achieved and as $p$ approaches 1, the taxicab/Manhattan norm is achieved. However, both of these values for $p$ result in constraint boundaries that are not continuously differentiable which is required for this reachability algorithm. Fortunately, these values of $p$ can be closely approximated with large (> 5) or small (< 1.5) values of $p$ which remain continuously differentiable.

In addition to specifying the value of $p$ to use for the p-norm constraint, there are three methods for defining a constraint size and orientation:

1. M - Transformation matrix from ellipsoid to sphere
2. E - Ellipsoid shape matrix
3. limits - Limits along each axes (symmetric about center)

For $M$, $\tilde{x} = M x$ where $\|\tilde{x}\|_p \leq 1$, $M > 0$, and $M^{-1}$ describes transformation matrix from sphere to ellipsoid.

$M^T$ matrix should have QR factorization with R diagonal meaning that M should consist only of rotation and scaling (no shear, etc).

For $E$, $x^T E x = x^T M^T M x \leq 1$ where $E = M^T M$

For limits, each ith entry gives the maximum distance from the center to the boundary of the constraint region. When limits is used, the $M$ matrix is diagonal, $limits = diag(M^{-1})$, and the resulting constraint region is aligned with the coordinate axes.

The default value of $p$ is 2, corresponding to sphere and ellipsoidal constraint regions. As $p$ is increased from 2, the constraint more closely resembles a rectangle (like inflating a balloon and seeing it fill a box).
Dynamic system classes are used by the Reachability object to propagate states, costates, flow states, and flow state transition matrices. There are multiple predefined dynamic system classes that can be created with ease:

1. Linear Time-invariant (LTI)
2. Linear Time-varying (LTV)
3. Sympy-Defined with Smooth Switching Approximations
4. Sympy-Defined with Exact Switching Functions
5. quasi Linear Parameter-varying (qLPV)

The LTI Dynamics have the following system model:
\[ \dot{x} = Ax + Bu \]

The LTV Dynamics have the following system model:
\[ \dot{x} = A(t)x + B(t)u \]

The Sympy-Defined Dynamics have the following system model:
\[ \dot{x} = f(x, u, t) \] in general. However, because the optimal control law must be computed analytically, the following system model is preferred as the optimal control law can be analytically computed (refer to Optimal Control Overview):
\[ \dot{x} = f_1(x, t) + f_2(x, t)u \]

The qLPV Dynamics have the following system model:
\[ \dot{x} = A(v) \ast (x - x_v(v)) + B(v)(u - u_v(v)) - h(e_v(v), x_v(v)) + H(x) \]

where \( v(x) \) denotes a scalar parameter that depends on the state and parametrizes other terms in the dynamics model, \( x_v(v) \) denotes a trim state, \( u_v(v) \) denotes a trim control input, and \( e_v(v) \) denotes other quantities that depend on \( v \).

All of the dynamic system classes are built in a way so they have the same forms of the methods/functions. This is so the reachability is able to swap out dynamics models without changing any of the internal code. As a result, each dynamic system class is required to have the following methods/functions with these arguments:

- \( dxdt(y,t) \)
- \( dpdt(y,t) \)
- \( dPhidt(y,t) \)
- \( dydt(y,t) \)
- \( dYdt(Y,t) \)
- \( H(y,t) \)
In each of the above functions/methods, \( y \) is the trajectory flow state defined by concatenating the state and costate \( y = [x, p] \). There is also the augmented flow state defined by concatenating the trajectory flow state with a vectorized form of its state transition matrix \( Y = [y, \text{vec}(\Phi)] \). The current time value is given by \( t \).

If you are dealing with a dynamic system that is not predefined, there are primarily two methods to create a dynamic system class to use in the reachability analysis.

1. Create a dynamics class of your own that contains the methods/functions and arguments listed above
2. Define the dynamic system model symbolically using Sympy, autocode the Sympy functions, then use either of the predefined Sympy dynamics classes.

There are templates and examples for both of these approaches given in the examples.
8.1 Reachability Module

class reach.Reachability(dynamics, ICconstraint, jsonLoadFileName=None)
    Bases: object
    Class for Reachability Analysis
    Compute samples of reachable sets and tubes using continuation methods

    particles
        List of particle objects. Each index of this particle list corresponds to the ID number of the particle.
        Type list

    neighborsLedger
        List of unique edges for entire particle mesh/graph/network. Every index of this list contains the list
        [pi_ID, pj_ID] where pi_ID and pj_ID correspond to the particle ID numbers (index of self.particles list)
        This is useful when performing iterations over every neighboring pair of particles.
        Type list

    neighborsList
        List of neighborhoods per particle. Every ith index of this list contains the list of particle ID numbers of
        the neighbors of particle i ([pi_Neighbor_ID1, pi_Neighbor_ID2, pi_Neighbor_ID3, ...])
        Type list

    numParticles
        Current number of particles in the mesh/graph for this reachability analysis.
        Type int

    squaredCost
        Boolean on whether or not to use the squared inner product performance metric as optimal control objective
        function or the inner product
        Type bool

    subspaceDim
        This specifies the dimension of the subspace of interest in the reachability analysis. This essentially defines
        the dimension that the particles are sampled in.
        Type int, subspaceDim <= N

    __init__(dynamics, ICconstraint, jsonLoadFileName=None)
        Constructor for reachability class.

    Parameters
• **dynamics** (*Dynamics Object*) – Dynamics object definition representing the dynamics for this reachability analysis

• **ICconstraint** (*Constraint Object*) – Constraint object representing the initial condition manifold for this reachability analysis

• **jsonLoadFileName** (*str, optional*) – Filename or path to json file that keeps the options for this reachability analysis

`initializeReach(subspaceDim=None, sampType='uniform', uniformSamplingRate=20)`

Initializes particle/support vector distribution based off of uniform distribution

**NOTICE** - This software only computes subspace reachability from the first `subspaceDim` number of state space components. If you want to compute a subspace reachable volume for a particle component/subspace of the state, the state vector and dynamics should be rearranged so the “important” states are listed first.

**Parameters**

• **subspaceDim** (*int, optional*) – Number of dimensions for the subspace of interest in the reachability analysis.

  If `subspaceDim` is None, the default value is `self.N` for a full state space reachability analysis

**NOTICE** - This software only computes subspace reachability from the first `subspaceDim` number of state space components. If you want to compute a subspace reachable volume for a particle component/subspace of the state, the state vector and dynamics should be rearranged so the “important” states are listed first.

• **sampType** (*{'uniform', 'box', 'octagonal'}, optional*) – String specifying the desired particle sampling technique within the subspace of interest (specified by `subspaceDim`)

  Options are:
  – ‘uniform’ - Equally spaced samples on sphere
  – ‘box’ - Bounding box samples. This creates a sample along both the positive and negative coordinate directions of each specified coordinate axis of the subspace (e.g. -x, +x, -y, +y). Provides 2*`subspaceDim` samples total
  – ‘octagonal’ - Provides 2*(`subspaceDim` ^2) samples total based on octagonal sampling. Octagonal sampling has box samples plus it also provides the bisection sample between each pair of box samples.

• **uniformSamplingRate** (*int, optional*) – Desired number of particles per (0, 2 \(\pi\)) dimension of angular coordinates vector

  When `subspaceDim` is 2, this argument equivalent to the total number of particles

  This parameter will only be used if the `sampType` = ‘uniform’

`computeReach(Tvec, contMethodOption=2, printProgress=True, particleIndex=None, newtonsCorrectionBool=False)`

Perform continuation/homotopy over time horizon for each particle to get reach object.

If `particleIndex` is None, continuation/homotopy is computed for each particle in `self.particles` list.

**Parameters**

• **Tvec** (*array_like*) – 1D array of time values for the reachability analysis. The first value should correspond to the initial time when the initial condition constraint is defined and the final time should correspond to the desired time horizon to compute the reach object
• **contMethodOption**(int) – Integer specifying which homotopy/continuation method is desired.
  
  Options are:
  
  1. Basic continuation method
  2. Pseudo-arclength continuation method
  3. Pseudo-arclength homotopy using Sard’s theorem (NOT WORKING)
  4. Predictor-Corrector with Newton step updates (IN DEVELOPMENT)
  5. Predictor-Corrector with Broyden step updates (IN DEVELOPMENT)

• **printProgress**(bool, optional) – Boolean determining whether or not a progress bar should be shown in the console/terminal while the continuation method is being performed.

• **particleIndex**(int, list of ints, None), optional) – Integer indices of self.particles list that method should be performed on.

• **newtonsCorrectionBool**(bool, optional) – Boolean determining whether or not Newton’s correction steps should be performed at the end of every continuation method iteration over time horizon.

**computeCurrentStateTrajectories**(numTimeSteps, particleIndex=None)

Computes current (at current time horizon, T) optimal trajectories as a function of t (x(t), p(t) from t = t0...T)

If dealing with a reach tube (where particles may be frozen), instead this computes the frozen trajectory (x(t), p(t) from t = t0...Tfr).

If **particleIndex** is None, trajectories are computed for each particle in self.particles list.

**Parameters**

• **numTimeSteps**(int) – number of time values (t) for the trajectories. This is like a time resolution for trajectories for plotting.

• **particleIndex**(int, list of ints, None), optional) – Integer indices of self.particles list that trajectories should be computed for.

**computeFinalStateHistories**(particleIndex=None)

Computes final states and costates as a function of time horizon, T.

This will create/update the particle.xf_T and particle.pf_T properties.

The size of particle.xf_T and particle.pf_T are N x numTimeSteps where N is dimension of state/costate and numTimeSteps is number of entries in self.Tvec.

This should normally be called after the self.computeReach method.

If **particleIndex** is None, trajectories are computed for each particle. If **particleIndex** is set, only those particles (specified by the indices of self.particles list) will have the quantities computed.

**Parameters**

• **particleIndex**(int, list of ints, None), optional) – Integer indices of self.particles list that trajectories should be computed for.

**computeTrajectoriesOverT**(numTimeSteps, particleIndex=None)

At each time horizon (T), compute trajectory for x and p (x(t), p(t)) for each specified particle.

This method computes and creates the particle.y_t_T property where the size is given by 2*N x numTimeSteps x numTimeHorizonSteps where N is the dimension of state/costate, the input argument num-
TimeSteps denotes the number of time values, t, to compute the trajectories, and numTimeHorizonSteps is number of entries in self.Tvec.

This should normally be called after the self.computeReach method.

If particleIndex is None, trajectories are computed for each particle in self.particles list.

Parameters

- numTimeSteps (int) – number of time values (t) for the trajectories. This is like a time resolution for trajectories for plotting.

- particleIndex (int, list of ints, None, optional) – Integer indices of particles list that trajectories should be computed for

updateReach (T)

createBallTree()

createBallTree() creates ball tree for efficient nearest neighbor searches

addParticles (thArray, dsArrayBool=False)

Add particles specified by thArray/dsArray to overall particle list.

Depending on the input argument, dsArrayBool, thArray will be treated as an array of angle coordinates or search unit vectors.

Creates and uses ball tree for nearest neighbor search to define new particle’s neighbors.

Automatically calls methods to compute reachability and state/costate trajectories.

Parameters

- thArray ((numNewParticles, N-1), (numNewParticles, N)) – Array defining angular coordinates or unit vector search directions for the new particles to add to the analysis.

- dsArrayBool (bool, optional) – Boolean that determines how the input array is treated.
  - False - Input array is treated as an array of angular coordinates
  - True - Input array is treated as an array of unit vector search directions

redistributeParticles()

updateGraphDistanceCost()

updateDistanceCost()

meshRefinement (method='bisect', cost='J', q1=25, q3=75, maxRefinements=3)

Iterated version of singleMeshRefinement where it repeats the outlier test and adding particles until no more particles need to be added (based on outlier parameters).

Parameters

- method (['bisect', 'cubic'], optional) – String that determines which mesh refinement technique is used.
  - 'bisect': Bisection based mesh refinement is performed.
  - 'cubic': Cubic minimization mesh refinement is performed.
• **cost (\{'J', 'D'\}, optional)** – String that specifies which distance metric to use in determining outliers
  - 'J' : Weighted graph distance cost
  - 'D' : Euclidean distance
• **q1 (int, optional)** – Integer between 1 and 49 specifying the first quartile
• **q3 (int, optional)** – Integer between 51 and 99 specifying the third quartile
  The closer this number is to 50 the more uniform it attempts to become
• **maxRefinements (int, optional)** – Integer that specifies the maximum number of refinement iterations

**singleMeshRefinement (method='bisect', cost='J', q1=25, q3=75)**
Performs a single iteration of mesh refinement where it computes edge-distance cost outliers and attempts to remove them by spawning a new particle in that region of the reach object

**Parameters**

• **method (\{'bisect', 'cubic'\}, optional)** – String that determines which mesh refinement technique is used
  - 'bisect' : Bisection based mesh refinement is performed
  - 'cubic' : Cubic minimization mesh refinement is performed
• **cost (\{'J', 'D'\}, optional)** – String that specifies which distance metric to use in determining outliers
  - 'J' : Weighted graph distance cost
  - 'D' : Euclidean distance
• **q1 (int, optional)** – Integer between 1 and 49 specifying the desired first quartile value for the IQR outlier detection
• **q3 (int, optional)** – Integer between 51 and 99 specifying the desired third quartile value for the IQR outlier detection
  The closer this number is to 50 the more uniform it attempts to become

**reduceError_Newton (convTol=1e-06, maxIter=5)**
Use Newton’s method to improve accuracy (reduce necessary condition error) of reach set at current time horizon, T

**Parameters**

• **convTol (float, optional)** – Desired tolerance for necessary condition of optimality constraint satisfaction (particle.F())
• **maxIter (int, optional)** – Integer that specifies the maximum number of Newton’s method updates

**vertices (prop='xf_T', returnNormals=False)**
Converts the specified reachable volume point solutions to an array (numParticles x numDimensions) of vertices.

If specified, will additionally return the corresponding normal vectors as an array (numParticles x numDimensions)

**Parameters**
• **prop** ('xf_T , x'), optional) – String specifying which particle attribute/property to save

• **returnNormals** (bool, optional) – Boolean that determines whether or not the normal vectors are also returned

**Returns**

• **vertArr** (array_like, shape (numParticles, N)) – Array that contains the particles states (either xf_T or x) denoting the vertices of a graph/mesh

• **normalArr** (array_like, shape (numParticles, N)) – Array that contains the particles costates (either pf_T or p) denoting the surface normals of a graph/mesh

**vertices_over_T** (returnNormals=False)

Converts the specified reachable volume point solutions to an array (numParticles x numDimensions x numTimeHorizonSteps) of vertices.

If specified, will additionally return the corresponding normal vectors as an array (numParticles x numDimensions x numTimeHorizonSteps)

Make sure `self.computeFinalStateHistories` is called before this so the `self.xf_T` and `self.pf_T` are computed

**Parameters**

• **returnNormals** (bool, optional) – Boolean that determines whether or not the normal vectors are also returned

**Returns**

• **vertArr** (array_like, shape (numParticles, N, numTimeHorizonSteps)) – Array that contains the particles states (xf_T) denoting the vertices of a graph/mesh

• **normalArr** (array_like, shape (numParticles, N, numTimeHorizonSteps)) – Array that contains the particles costates (pf_T) denoting the surface normals of a graph/mesh

**saveFigures** (figHandles, saveFilename=None, filePathStr='./Figures/', dpi=600)

Saves specified figures as png files in specified file path

If `saveFilename` is a single string, this function will save all of the figure handles provided with ‘saveFilename_0.png’, ‘saveFilename_1.png’, ‘saveFilename_2.png’, etc.

If `saveFilename` is a list of strings, each index of this list will be the saved filename for the same index in the figure handles list.

**Parameters**

• **figHandles** (list) – List of figure handles that need to be saved

• **saveFilename** ((str, list of str), optional) – Filename(s) (without extension) to save figures

  If `saveFilename` is a single string, this function will save all of the figure handles provided with ‘saveFilename_0.png’, ‘saveFilename_1.png’, ‘saveFilename_2.png’, etc.

  If `saveFilename` is a list of strings, each index of this list will be the saved filename for the same index in the figure handles list.

• **filePathStr** (str, optional) – String specifying file path to save figures

• **dpi** ((None, int), optional) – The resolution in dots per inch. If `None`, defaults to 600.

**convertReachSetToTube** (returnNew=True)

Converts a reach set to a reach tube. If specified, a copy of the reach object will be made before the conversion and returned.
**Parameters** `returnNew(bool, optional)` – List of figure handles that need to be saved

**Returns**

- `rt` – If `returnNew` is true, then a copy of the original reach set object is created. The conversion to the reach tube is performed on the copy and returned.

If `returnNew` is false, the current reach set is converted to a reach tube

**Return type** `Reachability` object, optional

### 8.2 Particle Module

```python
class particle.Particle (th, ICconstraint, neighborsInd=None, squaredCost=False, t0=0.0)
Bases: object
```

Class for Reachability Particle

Each particle corresponds to a sample on the boundary of a reachability volume

- **`ds, th`**
  - `ds` is the unit vector search direction for optimal control performance metric and support function. `th` is the hyperspherical coordinate representation of `ds`
  
  **Type** array_like, shape (N,) and (N-1,)

- **`x, p, y`**
  - State `x`, costate `p`, and flow state `y'=[x, p]` at the current time horizon, `T`
  
  **Type** array_like, shape (N,) , (N,) , (2N,)

- **`z`**
  - Optimal initial solution vector `z=[x0, lam]` where `x0` is optimal initial state (at initial time, `t0`) that satisfies the initial condition constraint and `lam` is the corresponding lagrange multiplier to the initial condition constraint
  
  **Type** array_like, shape (N+1,)

- **`dFdz`**
  - Jacobian of necessary condition of optimality function `F` with respect to `z`
  
  **Type** array_like (N+1, N+1)

- **`dFdT`**
  - Jacobian of necessary condition of optimality function `F` with respect to time horizon `T`
  
  **Type** array_like (N+1,)

- **`xf_T, pf_T, z_T`**
  - Final state, final costate, and optimal initial solution as a function of time horizon. This corresponds to the histories (over time horizon) of the final state, final costate, and optimal initial solution. `numTimeHorizonSteps` corresponds to the number of time horizon `T` values. These quantities are related to how the reachable volume over time horizon, `T`
  
  **Type** array_like, shape (N,numTimeHorizonSteps) , (N,numTimeHorizonSteps) , (N+1,numTimeHorizonSteps)

- **`x_t, p_t`**
  - State and costate trajectories from initial time `t0` to the current time horizon `T`. `numTimeSteps` corresponds to the number of time interval values `t` at current time horizon `T`
  
  **Type** array_like, shape (N,numTimeSteps), (N,numTimeSteps)
**\( y_{\cdot \cdot T} \)**  
Array that stores all the state/costate trajectories over time interval (t) given different time horizon values, \( T \).  
**Type** array_like, shape (2N, numTimeSteps, numTimeHorizonSteps)

**\( \text{__init__}(th, ICconstraint, neighborsInd=None, squaredCost=False, t0=0.0) \)**  
Constructor for particle class.

**Parameters**

- **\( th \)** (array_like, shape (N-1,)) – 1D numpy array of size N-1 where N is the dimension of the state space corresponding to angular coordinates in hyperspherical coordinates.
- **\( ICconstraint \)** (Constraint object) – Constraint object representing the initial condition manifold for this reachability analysis.
- **\( neighborsInd \)** (list, optional) – List of particle indices that correspond to the neighbors of this particle.
- **\( squaredCost \)** (bool, optional) – If true, a squared inner product performance metric is used in the reachability analysis and if false an inner product performance metric is used.
- **\( t0 \)** (int, float, optional) – Initial time value. Time at which initial condition constraint is defined.

**classmethod fromds(ds, ICconstraint, neighborsInd=None, squaredCost=False, t0=0.0)**  
Alternate constructor for particle class using unit vector search direction.

**Parameters**

- **\( ds \)** (array_like, shape (N,)) – Unit vector in state space given by 1D numpy array of size N where N is the dimension of the state space.
- **\( ICconstraint \)** (Constraint object) – Constraint object representing the initial condition manifold for this reachability analysis.
- **\( neighborsInd \)** (list, optional) – List of particle indices that correspond to the neighbors of this particle.
- **\( squaredCost \)** (bool, optional) – If true, a squared inner product performance metric is used in the reachability analysis and if false an inner product performance metric is used.
- **\( t0 \)** (int, float, optional) – Initial time value. Time at which initial condition constraint is defined.

**updatez(newz)**  
Updates particle properties (z,y0,x0,p0,\( \lambda \)) based on new z value.

**Parameters**
\( newz \) (array_like, shape (N+1,)) – 1D numpy array denoting the new optimal initial condition for the optimal control/reachability problem.

**updateth(newth)**  
Updates particle properties (th,ds) based on new theta value.

**Parameters**
\( newth \) (array_like, shape (N-1,)) – 1D numpy array denoting the new vector of angular coordinates for this particle.

**updateds(newds)**  
Updates particle properties (th,ds) based on new ds (unit vector search direction) value.
Parameters `newds` *(array_like, shape (N,)) –* 1D numpy array denoting the new unit vector search direction for this particle

`updatey` *(newy)*

Updates particle properties \((y, x, p)\) based on new \(y\) (state concatenated with costate) value

Parameters `newy` *(array_like, shape (2N,)) –* 1D numpy array denoting the current flow state for this particle

`updateFinalStates_y` *(dynamics)*

Uses current `self.z` and `self.T` to compute \(x_f, p_f, y_f, F\)

Does not include state transition matrix (STM)/Phi propagation, \(dFdz\), or \(dFdT\) jacobians. To compute these terms as well, use `self.updateFinalStates`

Parameters `dynamics` *(Dynamics object) –* Dynamics object definition to propagate this particle with

`updateFinalStates` *(dynamics)*

Uses current `self.z` and `self.T` to compute \(x_f, p_f, y_f, F\) and other things that result

Parameters `dynamics` *(Dynamics object) –* Dynamics object definition to propagate this particle with

\(h\) *(x=None)*

Compute support function value for this particle

Parameters `x` *(array_like, optional) –* Input state to evaluate this particle’s support function using this particle’s unit vector search direction (ds).

If not provided, uses `self.x` from particle to compute it’s support function

Returns `hVal` – Support function value

Return type `{float, array}`

\(V\) *(x=None)*

Compute inner product performance value for this particle

Parameters `x` *(array_like, optional) –* Input state to evaluate this particle’s performance value using this particle’s unit vector search direction (ds).

If not provided, uses `self.x` from particle to compute it’s performance value

Returns `VVal` – Inner product performance value

Return type `{float, array}`

\(Vx\) *(x=None)*

Compute performance value jacobian for this particle

Parameters `x` *(array_like, optional) –* Input state to evaluate this particle’s performance value using this particle’s unit vector search direction (ds).

If not provided, uses `self.x` from particle to compute it’s performance value

Returns `VxVal` – Inner product performance jacobian value

Return type `array_like, shape (N,)`

\(Vxx\)

Compute performance value hessian for this particle. Right now this is independent of state input

Returns `VxxVal` – Inner product performance hessian value

Return type `array_like, shape (N,N)`
**dF_dth()**
Compute jacobian of optimality constraint, F, with respect to angular coordinates, th, for this particle.

**Returns**
- **dF_dthVal**: Optimality constraint jacobian with respect to particle angular coordinates

**Return type**
array_like, shape (N+1,N-1)

**dF_dds()**
Compute jacobian of optimality constraint, F, with respect to unit vector search direction, ds, for this particle.

**Returns**
- **dF_ddsVal**: Optimality constraint jacobian with respect to particle unit vector search direction

**Return type**
array_like, shape (N+1,N)

**F()**
Evaluate particle’s optimality constraint function

**Returns**
- **FVal**: Optimality constraint value

**Return type**
array_like, shape (N+1,)

**FNorm()**
Evaluate 2-norm of particle’s optimality constraint function

**Returns**
- **FNormVal**: Optimality constraint norm value

**Return type**
float

**computeGraphDistanceCost** *(particles, Q)*
Compute Laplacian (weighted graph distance) for this particle. It also updates the particle.J property of every particle

**Parameters**
- **particles** (list) – List of particle objects. Each index should be ordered so particle.neighborsInd is still true
- **Q** (array_like, shape (subspaceDim, N)) – Weighting matrix

**Returns**
- **JVal**: Laplacian (weighted graph distance) cost value for this particle

**Return type**
float

**computeDistanceCost** *(particles, subspaceDim=2)*
Compute Euclidean distance cost for this particle. It also updates the particle.D property of every particle

**Parameters**
- **particles** (list) – List of particle objects. Each index should be ordered so particle.neighborsInd is still true
- **subspaceDim** (int) – Number of dimensions in subspace of interest in reachability problem

**Returns**
- **DVal**: Euclidean distance cost value for this particle

**Return type**
float

### 8.3 Constraint Module

**class** `constraints.Constraint` *(M, p=2, xc=None, constType='F')*

**Bases:** object
Class for P-Norm Constraints

This class represents p-norm constraints and provides functions for evaluating useful quantities related to the constraint.

In addition to specifying the value of \( p \) to use for the p-norm constraint, there are three methods for defining a constraint size and orientation:

1. \( M \) - Transformation matrix from ellipsoid to sphere
2. \( E \) - Ellipsoid shape matrix
3. \( \text{limits} \) - Limits along each axes (symmetric about center)

For \( M, \tilde{x} = Mx \) where \( \|\tilde{x}\|_{p,F} \leq 1, M > 0 \), and \( M^{-1} \) describes transformation matrix from sphere to ellipsoid

\( M^T \) matrix should have QR factorization with R diagonal meaning that M should consist only of rotation and scaling (no shear, etc)

For \( E, x^TEx = x^TM^TMx \leq 1 \) where \( E = M^TM \)

For \( \text{limits} \), each \( i \)th entry gives the maximum distance from the center to the boundary of the constraint region. When \( \text{limits} \) is used, the \( M \) matrix is diagonal, \( \text{limits} = \text{diag}(M^{-1}) \), and the resulting constraint region is aligned with the coordinate axes.

The default value of \( p \) is 2, corresponding to sphere and ellipsoidal constraint regions. As \( p \) is increased from 2, the constraint more closely resembles a rectangle (like inflating a balloon and seeing it fill a box)

\[ \text{__init__}(M, p=2, xc=None, constType='F') \]

Constructor for constraint class.

Parameters

- \( M \) (array_like, shape (\( N,N \))) - Transformation matrix from ellipsoid to sphere
  - \( \tilde{x} = Mx \) where \( \|\tilde{x}\|_{p,F} \leq 1, M > 0 \), and \( M^{-1} \) describes transformation matrix from sphere to ellipsoid

\( M^T \) matrix should have QR factorization with R diagonal meaning that M should consist only of rotation and scaling (no shear, etc)

- \( p \) (int, float, \( p > 1 \)) - p norm value to use for this constraint

- \( xc \) (array_like, shape (\( N, \)), optional) - Center coordinate of control constraint. If not provided, the default value is the origin

- \( \text{constType} \) (\( \{'F', 'p'\}, \) optional) - Type of constraint to use based off of p-norm or F-norm unit ball

classmethod \text{fromShapeMatrix}(E, p=2, xc=None, constType='F')

Alternate constructor for constraint class using shape matrix \( \text{transpose}(M)^*M \)

Shape matrix based from ellipsoid equation, \( x^TEx = x^TM^TMx \leq 1 \)

Parameters

- \( E \) (array_like, shape (\( N,N \))) - Shape matrix where \( x^TEx = x^TM^TMx \leq 1 \)

- \( p \) (int, float, \( p > 1 \)) - p norm value to use for this constraint

- \( xc \) (array_like, shape (\( N, \)), optional) - Center coordinate of control constraint. If not provided, the default value is the origin

- \( \text{constType} \) (\( \{'F', 'p'\}, \) optional) - Type of constraint to use based off of p-norm or F-norm unit ball
**classmethod fromLimits** *(limits, p=2, xc=None, constType='F')*

Alternate constructor for constraint class using limits along each state/coordinate dimension

**Parameters**

- `limits` *(array_like, list, size N)* – 1D list or array that define the max deviation of the constraint from it’s center along each axis
- `p` *(int, float) , p > 1* – p norm value to use for this constraint
- `xc` *(array_like, shape (N,) , optional)* – Center coordinate of control constraint. If not provided, the default value is the origin
- `constType` *({'F', 'p'}, optional)* – Type of constraint to use based off of p-norm or F-norm unit ball

**g** *(x, constType=None)*

Evaluate constraint at specified state location. If negative, the state is within the constraint region and if positive, the state is outside of the constraint region.

Equivalent to \|x\|_{p,F} - 1

**Parameters**

- `x` *(array_like, shape (N,))* – State to evaluate the constraint
- `constType` *(str, optional)* – Constraint type. This argument specifies whether or not the constraint is based on p-norm/Minkowski distance metric ('p') or the F-norm equivalent of it ('F'). If not provided, defaults to constraint constraint type (self.constType)

**Returns**

- `gVal` – Constraint satisfaction value where gVal = 0 signifies that the state is on the boundary of the constraint region

If gVal is negative, the state is within the constraint region and if gVal is positive, the state is outside of the constraint region

**Return type** float

**Dg** *(x, constType=None)*

Evaluate constraint gradient at specified state location. Equivalent to d(\|x\|_{p,F})/dx

**Parameters**

- `x` *(array_like, shape (N,))* – State to evaluate the constraint gradient
- `constType` *(str, optional)* – Constraint type. This argument specifies whether or not the constraint is based on p-norm/Minkowski distance metric ('p') or the F-norm equivalent of it ('F'). If not provided, defaults to constraint constraint type (self.constType)

**Returns**

- `DgVal` – Constraint gradient vector at given state, x

**Return type** array_like, shape (N,)

**D2g** *(x, constType=None, eps=1e-06)*

Evaluate constraint hessian at specified state location. Equivalent to d2(\|x\|_{p,F})/dx2

If p >= 2, the exact solution will be computed and returned. However, for 1 < p < 2, this derivative doesn’t exist (it’s infinite). In these cases (1 < p < 2), an approximate value of the derivative will be returned based on smooth approximations based on the argument eps. As eps approches 0, the approximation becomes more accurate.
Parameters
• \( \mathbf{x} \) (array_like, shape (N,)) – State to evaluate the constraint hessian
• \( \text{constType} \) (\{'F', 'p'\}, str, optional) – Constraint type. This argument specifies whether or not the constraint is based on p-norm/Minkowski distance metric (‘p’) or the F-norm equivalent of it (‘F’). If not provided, defaults to constraint constraint type (self.constType)
• \( \epsilon \) (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As \( \epsilon \) approaches 0, the approximations approach the true functions when component of \( \mathbf{x} \) is near zero

Returns D2gVal – Constraint hessian matrix at given state, \( \mathbf{x} \)

Return type array_like, shape (N,N)

\( D3g(x, \text{constType}=None, \epsilon=1e-06) \)
Evaluate constraint triple derivative at specified state location. Equivalent to \( \frac{d^3||\mathbf{x}\|_{p,F}}{dx^3} \)
If \( p \geq 3 \) or \( p = 2 \), the exact solution will be computed and returned. However, for \( 1 < p < 2 \) and \( 2 > p > 3 \), this derivative doesn’t exist (it’s infinite). In these cases, an approximate value of the derivative will be returned based on smooth approximations based on the argument \( \epsilon \). As \( \epsilon \) approaches 0, the approximation becomes more accurate.

Parameters
• \( \mathbf{x} \) (array_like, shape (N,)) – State to evaluate the constraint triple derivative
• \( \text{constType} \) (\{'F', 'p'\}, str, optional) – Constraint type. This argument specifies whether or not the constraint is based on p-norm/Minkowski distance metric (‘p’) or the F-norm equivalent of it (‘F’). If not provided, defaults to constraint constraint type (self.constType)
• \( \epsilon \) (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As \( \epsilon \) approaches 0, the approximations approach the true functions when component of \( \mathbf{x} \) is near zero

Returns D3gVal – Constraint hessian matrix at given state, \( \mathbf{x} \)

Return type array_like, shape (N,N,N)

\( \text{randomSample}(\text{scaleFactor}=\text{None}) \)
Computes a random sample that lies on or within the constraint

Parameters scaleFactor (float, \( 0 < \text{scaleFactor} \leq 1 \), optional) – If scaleFactor isn’t provided or is equal to 1, the random sample will lie on the boundary of the constraint region. If scaleFactor is less than 1, the random sample will lie within the constraint region

Returns sample – Random sample that lies on or within the constraint region

Return type array_like, shape (N,)

\( \text{maxInnerProduct}(y, \text{returnLambda}=False, \text{constType}=\text{None}) \)
Returns the point \( \mathbf{x} \) that maximizes inner product between \( y \) and \( \mathbf{x} \) (\( \text{dot}(y,\mathbf{x}) \)) where \( \mathbf{x} \) satisfies the constraint.

Parameters
• \( \mathbf{y} \) (array_like, shape (N,)) – Input vector to maximize inner product over
• returnLambda (bool, optional) – Boolean to determine whether or not the corresponding lagrange multiplier to this problem should be returned
• **constType**: ('F', 'p'), str, optional – Constraint type. This argument specifies whether or not the constraint is based on \( p \)-norm/Minkowski distance metric ('p') or the F-norm equivalent of it ('F'). If not provided, defaults to constraint constraint type (self.constType)

**Returns**

• **x** (array_like, shape (N,)) – Vector that maximizes inner product with \( y \) and also satisfies constraint

• **lam** (float) – Lagrange multiplier for this constrained optimization problem

**maxInnerProductSmooth** \((y, eps=1e-06)\)

Returns the point \( x \) that maximizes inner product between \( y \) and \( x \) (\( \text{dot}(y,x) \)) where \( x \) satisfies the constraint.

Uses smooth approximations of sign(x) and abs(x) depending on the value of \( eps \)

**Parameters**

• **y** (array_like, shape (N,)) – Input vector to maximize inner product over

• **eps** (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As \( eps \) approaches 0, the approximations approach the true functions when \( x \) is near zero

**Returns** x – Vector that maximizes inner product with \( y \) and also satisfies constraint

**Return type** array_like, shape (N,)

**maxInnerProductSmoothJacobian** \((y, eps=1e-06)\)

Returns the jacobian \( dx/dy \) where \( y/x \) are the input/output to the **maxInnerProduct** function.

Uses smooth approximations of sign(x) and abs(x) depending on the value of \( eps \)

**Parameters**

• **y** (array_like, shape (N,)) – Input vector to maximize inner product over

• **eps** (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As \( eps \) approaches 0, the approximations approach the true functions when \( x \) is near zero

**Returns** \( dxdy \) – Jacobian of constraint maximal inner product function with respect to the input vector

**Return type** array_like, shape (N,N)

**maxInnerProductSmoothJacobian2** \((y, eps=1e-06)\)

Returns the jacobian \( dx/dy \) where \( y/x \) are the input/output to the **maxInnerProduct** function.

Uses slightly different computations compared to **maxInnerProductSmoothJacobian**

Uses smooth approximations of sign(x) and abs(x) depending on the value of \( eps \)

**Parameters**

• **y** (array_like, shape (N,)) – Input vector to maximize inner product over

• **eps** (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As \( eps \) approaches 0, the approximations approach the true functions when \( x \) is near zero

**Returns** \( dxdy \) – Jacobian of constraint maximal inner product function with respect to the input vector
**Return type** array_like, shape (N,N)

**maxInnerProductJacobianNumerical** (*y, h=0.0001*)
Uses central finite differences to approximate the jacobian dx/dy where y/x are the input/output to the maxInnerProduct function.

**Parameters**
- **y** (array_like, shape (N,)) – Input vector to maximize inner product over
- **h** (float, optional) – Scalar determining the step size in the central finite differencing

**Returns** dx dy – Jacobian of constraint maximal inner product function with respect to the input vector

**Return type** array_like, shape (N,N)

**smoothJacobianTensor** (*y, eps=1e-06*)
Returns the jacobian d/dy(dx/dy) where y/x are the input/output to the maxInnerProduct function.

Uses smooth approximations of sign(x) and abs(x) depending on the value of eps

**Parameters**
- **y** (array_like, shape (N,)) – Input vector to maximize inner product over
- **eps** (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As eps approaches 0, the approximations approach the true functions when x is near zero

**Returns** dx dy dy – Double Jacobian (tensor) of constraint maximal inner product function with respect to the input vector

**Return type** array_like, shape (N,N,N)

**constraints.pNormJacobian** (*y, p*)
Returns the jacobian d(||y||_p)/dy where y is an input vector and p is the value of the p-norm

Uses smooth approximations of sign(x) and abs(x) depending on the value of eps

**Parameters**
- **y** (array_like, shape (N,)) – Input vector to maximize inner product over
- **eps** (float, optional) – Scalar determining the degree of sharpness in the smooth approximations. As eps approaches 0, the approximations approach the true functions when x is near zero

**Returns** dx dy – Jacobian of constraint maximal inner product function with respect to the input vector

**Return type** array_like, shape (N,N)

### 8.4 Dynamic System Module

**class** dynSystems.LTIDynamics (*A, B, uLimit=1, pNorm=2, signApproxEps=1e-06*)

**Bases:** object

Class for Linear Time Invariant (LTI) dynamic systems of the form

\[ xDot = A*x + B*u \]

where A is n x n, B is n x m and u control input has p-norm type constraint
__init__ (A, B, uLimit=1, pNorm=2, signApproxEps=1e-06)
Constructor for LTI dynamics system class.

Parameters
- **A** (*array_like*, shape *(N,N)*) – State space system matrix
- **B** (*array_like*, shape *(N,M)*) – State space control input matrix
- **uLimit** (*{array_like, float, int, list}*, optional) – Max control input along each dimension of control vector
- **pNorm** (*{float, int}*), optional) – p-norm value for constraint
- **signApproxEps** (*float*, optional) – Scalar determining the degree of sharpness in the smooth approximations of sign() and abs()

As signApproxEps approaches 0, the approximations approach the true functions

**uStar** *(y, t)*
Computes optimal control *(u)* given flow state *(y = [x, p])* where x is state and p is costate

Parameters
- **y** (*array_like*, shape *(2N,)*) – Flow state, y = [x, p], where x is state and p is costate
- **t** (*float*) – Time value

Returns **uStar** – Optimal control input given current flow state and time

Return type array_like, shape *(M,)*

**dxdt** *(y, t)*
Computes state dynamics *(dx/dt)* given flow state *(y = [x, p])* where x is state and p is costate

Parameters
- **y** (*array_like*, shape *(2N,)*) – Flow state, y = [x, p], where x is state and p is costate
- **t** (*float*) – Time value

Returns **xd** – State dynamics given current flow state and time

Return type array_like, shape *(N,)*

**dpdt** *(y, t)*
Computes costate dynamics *(dp/dt)* given flow state *(y = [x, p])* where x is state and p is costate

Parameters
- **y** (*array_like*, shape *(2N,)*) – Flow state, y = [x, p], where x is state and p is costate
- **t** (*float*) – Time value

Returns **pd** – Costate dynamics given current flow state and time

Return type array_like, shape *(N,)*

**dPhidt** *(Y, t)*
Computes state transition matrix STM dynamics *(dPhi/dt)* given flow state *(Y = [x, p, Phi])* where x is state, p is costate, and Phi is STM for y

Parameters
• $Y$ (*array_like, shape $(2N + (2N)^2)$*) – Augmented flow state, $Y = [x, p, \text{vec(Phi)}]$, where $x$ is state, $p$ is costate, and $\text{Phi}$ is STM for $y$

• $t$ (*float*) – Time value

**Returns** Phid – State dynamics given current flow state and time

**Return type** *array_like, shape $(2N,2N)$*

**$H(y, t)$**
Computes optimal control Hamiltonian given flow state ($y = [x, p]$) where $x$ is state and $p$ is costate

**Parameters**

• $y$ (*array_like, shape $(2N,)$*) – Flow state, $y = [x, p]$, where $x$ is state and $p$ is costate

• $t$ (*float*) – Time value

**Returns** HVal – Optimal control Hamiltonian value given current flow state and time

**Return type** *float*

**$dydt(y, t)$**
Computes trajectory flow dynamics ($dy/dt$) given flow state ($y = [x, p]$) where $x$ is state and $p$ is costate

**Parameters**

• $y$ (*array_like, shape $(2N,)$*) – Flow state, $y = [x, p]$, where $x$ is state and $p$ is costate

• $t$ (*float*) – Time value

**Returns** yd – Flow dynamics given current flow state and time

**Return type** *array_like, shape $(2N,)$*

**$dYdt(Y, t)$**
Computes augmented flow dynamics ($d\Phi/dt$) given flow state ($Y = [x, p, \text{Phi}]$) where $x$ is state, $p$ is costate, and $\text{Phi}$ is STM for $y$

**Parameters**

• $Y$ (*array_like, shape $(2N + (2N)^2,)$*) – Augmented flow state, $Y = [x, p, \text{vec(Phi)}]$, where $x$ is state, $p$ is costate, and $\text{Phi}$ is STM for $y$

• $t$ (*float*) – Time value

**Returns** Yd – Augmented flow dynamics given current flow state and time

**Return type** *array_like, shape $(2N + (2N)^2)$*

**propagate_y(y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)**
Propagate flow state $y0 = [x0, p0]$ from $t0$ to $yf = [xf, pf]$ at $tf$

**Parameters**

• $y0$ (*array_like, shape $(2N,)$*) – Initial flow state, $y0 = [x0, p0]$, where $x0$ is initial state and $p0$ is initial costate

• $t0$ (*float*) – Initial time value

• $tf$ (*float*) – Final time value

• $\text{numTimeSteps}$ (*int, optional*) – If not provided, this function will only return $yf = [xf, pf]$

---

**8.4. Dynamic System Module**

39
If provided, this function will return \( x(t), p(t) \) with \( t \) from \( t_0 \) to \( t_f \) with \textit{numTimeSteps} time values

- \texttt{atol} (\texttt{rtol},) – Relative and absolute tolerance for the differential equation solver

\textbf{Returns} \( y_f \) – Depending on the input argument of \textit{numTimeSteps}, this will either be the final flow state \( y_f = [x_f, p_f] \) or it would be an array with \( y(t) \) from \( t_0 \) to \( t_f \) with \textit{numTimeSteps} time values

\textbf{Return type} \ array\_like, shape (2N,) or (\textit{numTimeSteps},2N)

\texttt{propagate\_Y} \( (y_0, t_0, t_f, \textit{numTimeSteps} = \text{None}, \texttt{rtol} = 1e-06, \texttt{atol} = 1e-06) \)

Propagate augmented flow state \( Y_0 = [x_0, p_0, \text{vec}(\Phi_0)] \) from \( t_0 \) to \( Y_f = [x_f, p_f, \text{vec}(\Phi_f)] \) at \( t_f \)

\textbf{Parameters}

- \texttt{y0} (\texttt{array\_like}, shape (2N,)) – Initial flow state, \( y_0 = [x_0, p_0] \), where \( x_0 \) is initial state and \( p_0 \) is initial costate
- \texttt{t0} (\texttt{float}) – Initial time value
- \texttt{tf} (\texttt{float}) – Final time value
- \texttt{numTimeSteps} (\texttt{int}, \texttt{optional}) – If not provided, this function will only return \( Y_f = [x_f, p_f, \text{vec}(\Phi_f)] \)

If provided, this function will return \( x(t), p(t) \) with \( t \) from \( t_0 \) to \( t_f \) with \textit{numTimeSteps} time values

- \texttt{atol} (\texttt{rtol},) – Relative and absolute tolerance for the differential equation solver

\textbf{Returns} \( Y_f \) – Depending on the input argument of \textit{numTimeSteps}, this will either be the final augmented flow state \( Y_f = [x_f, p_f, \text{vec}(\Phi_f)] \) or it would be an array with \( Y(t) \) from \( t_0 \) to \( t_f \) with \textit{numTimeSteps} time values

\textbf{Return type} \ array\_like, shape (2N + (2N)^2,) or (\textit{numTimeSteps},2N + (2N)^2)

\texttt{class} dynSystems.LTVDynamics \( (A, B, uLimit=1, pNorm=2, t0=0.0, \text{signApproxEps}=1e-06) \)

\textbf{Bases:} \texttt{object}

Class for Linear Time-Varying (LTV) dynamic systems of the form

\( xDot = A(t)x + B(t)u \)

where \( A \) is \( n \times n \), \( B \) is \( n \times m \) and \( u \) control input has p-norm type constraint

\texttt{__init__} \( (A, B, uLimit=1, pNorm=2, t0=0.0, \text{signApproxEps}=1e-06) \)

Constructor for LTI dynamics system class.

\textbf{Parameters}

- \texttt{A} (\texttt{array\_like}, shape (\( N, N \))) – State space system matrix
- \texttt{B} (\texttt{array\_like}, shape (\( N, M \))) – State space control input matrix
- \texttt{uLimit} (\texttt{array\_like}, \texttt{float}, \texttt{int}, \texttt{list}, \texttt{optional}) – Max control input along each dimension of control vector
- \texttt{pNorm} (\texttt{float}, \texttt{int}, \texttt{optional}) – p-norm value for constraint
- \texttt{signApproxEps} (\texttt{float}, \texttt{optional}) – Scalar determining the degree of sharpness in the smooth approximations of \( \text{sign()} \) and \( \text{abs()} \)

As \( \text{signApproxEps} \) approaches 0, the approximations approach the true functions

\texttt{uStar} \( (y, t) \)

Computes optimal control \( (u) \) given flow state \( (y = [x, p]) \) where \( x \) is state and \( p \) is costate
Parameters

- **y (array_like, shape (2N,))** – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- **t (float)** – Time value

**Returns uStar** – Optimal control input given current flow state and time

**Return type** array_like, shape (M)

**dxdt (y, t)**
Computes state dynamics \((dx/dt)\) given flow state \((y = [x, p])\) where \(x\) is state and \(p\) is costate

Parameters

- **y (array_like, shape (2N,))** – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- **t (float)** – Time value

**Returns xd** – State dynamics given current flow state and time

**Return type** array_like, shape (N)

**dpdt (y, t)**
Computes costate dynamics \((dp/dt)\) given flow state \((y = [x, p])\) where \(x\) is state and \(p\) is costate

Parameters

- **y (array_like, shape (2N,))** – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- **t (float)** – Time value

**Returns pd** – Costate dynamics given current flow state and time

**Return type** array_like, shape (N)

**dPhidt (Y, t)**
Computes state transition matrix STM dynamics \((dPhi/dt)\) given flow state \((Y = [x, p, Phi])\) where \(x\) is state, \(p\) is costate, and \(Phi\) is STM for \(y\)

Parameters

- **Y (array_like, shape (2N + (2N)^2,))** – Augmented flow state, \(Y = [x, p, \text{vec}(Phi)]\), where \(x\) is state, \(p\) is costate, and \(Phi\) is STM for \(y\)
- **t (float)** – Time value

**Returns Phid** – State dynamics given current flow state and time

**Return type** array_like, shape (2N,2N)

**H (y, t)**
Computes optimal control Hamiltonian given flow state \((y = [x, p])\) where \(x\) is state and \(p\) is costate

Parameters

- **y (array_like, shape (2N,))** – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- **t (float)** – Time value

**Returns HVAl** – Optimal control Hamiltonian value given current flow state and time

**Return type** float

8.4. Dynamic System Module

41
\texttt{dydt} \ (y, t)  
Computes trajectory flow dynamics (dy/dt) given flow state \(y = [x, p]\) where \(x\) is state and \(p\) is costate

**Parameters**
- \(y\) (array_like, shape \((2N,)\)) – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- \(t\) (float) – Time value

**Returns**  
\(\text{yd}\) – Flow dynamics given current flow state and time

**Return type**  
array_like, shape \((2N,)\)

\texttt{dYdt} \ (Y, t)  
Computes augmented flow dynamics (dPhi/dt) given flow state \(Y = [x, p, \Phi]\) where \(x\) is state, \(p\) is costate, and \(\Phi\) is STM for \(y\)

**Parameters**
- \(Y\) (array_like, shape \((2N + (2N)^2,)\)) – Augmented flow state, \(Y = [x, p, \text{vec}(\Phi)]\), where \(x\) is state, \(p\) is costate, and \(\Phi\) is STM for \(y\)
- \(t\) (float) – Time value

**Returns**  
\(\text{Yd}\) – Augmented flow dynamics given current flow state and time

**Return type**  
array_like, shape \((2N + (2N)^2,)\)

\texttt{propagate\_y} \ (y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)  
Propagate flow state \(y0 = [x0, p0]\) from \(t0\) to \(yf = [xf, pf]\) at \(tf\)

**Parameters**
- \(y0\) (array_like, shape \((2N,)\)) – Initial flow state, \(y0 = [x0, p0]\), where \(x0\) is initial state and \(p0\) is initial costate
- \(t0\) (float) – Initial time value
- \(tf\) (float) – Final time value
- \(\text{numTimeSteps}\) (int, optional) – If not provided, this function will only return \(yf = [xf, pf]\)
  
  If provided, this function will return \(x(t), p(t)\) with \(t\) from \(t0\) to \(tf\) with \(\text{numTimeSteps}\) time values
- \(\text{atol}\) (\(\text{rtol}\)) – Relative and absolute tolerance for the differential equation solver

**Returns**  
\(yf\) – Depending on the input argument of \(\text{numTimeSteps}\), this will either be the final flow state \(yf = [xf, pf]\) or it would be an array with \(y(t)\) from \(t0\) to \(tf\) with \(\text{numTimeSteps}\) time values

**Return type**  
array_like, shape \((2N,)\) or \((\text{numTimeSteps}, 2N)\)

\texttt{propagate\_Y} \ (y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)  
Propagate augmented flow state \(Y0 = [x0, p0, \text{vec}(\Phi0)]\) from \(t0\) to \(Yf = [xf, pf, \text{vec}(\Phi f)]\) at \(tf\)

**Parameters**
- \(y0\) (array_like, shape \((2N,)\)) – Initial flow state, \(y0 = [x0, p0]\), where \(x0\) is initial state and \(p0\) is initial costate
- \(t0\) (float) – Initial time value
- \(tf\) (float) – Final time value
• **numTimeSteps**(int, optional) – If not provided, this function will only return Yf = [xf, pf, vec(Phif)]

If provided, this function will return x(t), p(t) with t from t0 to tf with numTimeSteps time values

• **atol**(rtol,) – Relative and absolute tolerance for the differential equation solver

**Returns Yf** – Depending on the input argument of numTimeSteps, this will either be the final augmented flow state Yf = [xf, pf, vec(Phif)] or it would be an array with Y(t) from t0 to tf with numTimeSteps time values

**Return type** array_like, shape (2N + (2N)^2,) or (numTimeSteps,2N + (2N)^2)

class dynSystems.SympyDynamicsSmooth(n, fname)

Bases: object

Class for Sympy Dynamic Systems with Smooth Approximations of sign(), abs(), optimal control

**__init__**(n, fname)

Constructor for Smooth Sympy Dynamics class

**Parameters**

• **n**(int) – Dimension/size of state

• **fname**(str) – denoting filename within tempFiles folder - usually has the .modname file extension

dydt**(y, t)**

Computes trajectory flow dynamics (dy/dt) given flow state (y = [x, p]) where x is state and p is costate

**Parameters**

• **y**(array_like, shape (2N,)) – Flow state, y = [x, p], where x is state and p is costate

• **t**(float) – Time value

**Returns yd** – Flow dynamics given current flow state and time

**Return type** array_like, shape (2N,)

dYdt**(Y, t)**

Computes augmented flow dynamics (dPhi/dt) given flow state (Y = [x, p, Phi]) where x is state, p is costate, and Phi is STM for y

**Parameters**

• **Y**(array_like, shape (2N + (2N)^2,)) – Augmented flow state, Y = [x, p, vec(Phif)], where x is state, p is costate, and Phi is STM for y

• **t**(float) – Time value

**Returns Yd** – Augmented flow dynamics given current flow state and time

**Return type** array_like, shape (2N + (2N)^2,)

dxdt**(y, t)**

Computes state dynamics (dx/dt) given flow state (y = [x, p]) where x is state and p is costate

**Parameters**

• **y**(array_like, shape (2N,)) – Flow state, y = [x, p], where x is state and p is costate

• **t**(float) – Time value
Returns **xd** – State dynamics given current flow state and time

Return type **array_like**, shape (N,)

**dpdt** *(y, t)*

Computes costate dynamics \(\frac{dp}{dt}\) given flow state \(y = [x, p]\) where \(x\) is state and \(p\) is costate

Parameters

- **y** *(array_like, shape (2N,)) – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- **t** *(float) – Time value

Returns **pd** – Costate dynamics given current flow state and time

Return type **array_like**, shape (N,)

**dPhidt** *(Y, t)*

Computes state transition matrix STM dynamics \(\frac{d\Phi}{dt}\) given flow state \(Y = [x, p, \Phi]\) where \(x\) is state, \(p\) is costate, and \(\Phi\) is STM for \(y\)

Parameters

- **Y** *(array_like, shape (2N + (2N)^2,)) – Augmented flow state, \(Y = [x, p, \text{vec}(\Phi)]\), where \(x\) is state, \(p\) is costate, and \(\Phi\) is STM for \(y\)
- **t** *(float) – Time value

Returns **Phid** – State dynamics given current flow state and time

Return type **array_like**, shape (2N,2N)

**H** *(y, t)*

Computes optimal control Hamiltonian given flow state \(y = [x, p]\) where \(x\) is state and \(p\) is costate

Parameters

- **y** *(array_like, shape (2N,)) – Flow state, \(y = [x, p]\), where \(x\) is state and \(p\) is costate
- **t** *(float) – Time value

Returns **HVal** – Optimal control Hamiltonian value given current flow state and time

Return type **float**

**propagate_y** *(y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)*

Propagate flow state \(y0 = [x0, p0]\) from \(t0\) to \(yf = [xf, pf]\) at \(tf\)

Parameters

- **y0** *(array_like, shape (2N,)) – Initial flow state, \(y0 = [x0, p0]\), where \(x0\) is initial state and \(p0\) is initial costate
- **t0** *(float) – Initial time value
- **tf** *(float) – Final time value
- **numTimeSteps** *(int, optional) – If not provided, this function will only return \(yf = [xf, pf]\)*

If provided, this function will return \(x(t), p(t)\) with \(t\) from \(t0\) to \(tf\) with \(numTimeSteps\) time values

- **atol** *(rtol,) – Relative and absolute tolerance for the differential equation solver*
returns yf – Depending on the input argument of numTimeSteps, this will either be the final flow state yf = [xf, pf] or it would be an array with y(t) from t0 to tf with numTimeSteps time values

return type array_like, shape (2N,) or (numTimeSteps,2N)

propagate_Y (y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)
Propagate augmented flow state Y0 = [x0, p0, vec(Phi0)] from t0 to Yf = [xf, pf, vec(Phif)] at tf

Parameters

• y0 (array_like, shape (2N,)) – Initial flow state, y0 = [x0, p0], where x0 is initial state and p0 is initial costate
• t0 (float) – Initial time value
• tf (float) – Final time value
• numTimeSteps (int, optional) – If not provided, this function will only return Yf = [xf, pf, vec(Phif)]

If provided, this function will return x(t),p(t) with t from t0 to tf with numTimeSteps time values
• atol (rtol,) – Relative and absolute tolerance for the differential equation solver

returns Yf – Depending on the input argument of numTimeSteps, this will either be the final augmented flow state Yf = [xf, pf, vec(Phif)] or it would be an array with Y(t) from t0 to tf with numTimeSteps time values

return type array_like, shape (2N + (2N)^2,) or (numTimeSteps,2N + (2N)^2)

class dynSystems.SympyDynamicsSwitch (n, fname)

Bases: object

Class for Sympy Dynamic Systems with Switching Functions defined

__init__ (n, fname)
Constructor for Smooth Sympy Dynamics class

Parameters

• n (int) – Dimension/size of state
• fname (str) – denoting filename within tempFiles folder - usually has the .modname file extension

dydt (y, t)
Computes trajectory flow dynamics (dy/dt) given flow state (y = [x, p]) where x is state and p is costate

Parameters

• y (array_like, shape (2N,)) – Flow state, y = [x, p], where x is state and p is costate
• t (float) – Time value

Returns yd – Flow dynamics given current flow state and time

Return type array_like, shape (2N,)

dYdt (Y, t)
Computes augmented flow dynamics (dPhi/dt) given flow state (Y = [x, p, Phi]) where x is state, p is costate, and Phi is STM for y

Parameters
• \( \mathbf{Y} (\text{array_like}, \text{shape} \ (2N + (2N)^2,)) \) – Augmented flow state, \( \mathbf{Y} = [x, p, \text{vec(Phi)}] \), where \( x \) is state, \( p \) is costate, and \( \Phi \) is STM for \( y \)

• \( t \ (\text{float}) \) – Time value

**Returns** \( \mathbf{Yd} \) – Augmented flow dynamics given current flow state and time

**Return type** array_like, shape \((2N + (2N)^2,)\)

\( \text{dxdt}(\mathbf{y}, t) \)
Computes state dynamics \( \frac{dx}{dt} \) given flow state \( (y = [x, p]) \) where \( x \) is state and \( p \) is costate

**Parameters**
- \( \mathbf{y} \ (\text{array_like, shape} \ (2N,)) \) – Flow state, \( y = [x, p] \), where \( x \) is state and \( p \) is costate
- \( t \ (\text{float}) \) – Time value

**Returns** \( \mathbf{xd} \) – State dynamics given current flow state and time

**Return type** array_like, shape \((N,)\)

\( \text{dpdt}(\mathbf{y}, t) \)
Computes costate dynamics \( \frac{dp}{dt} \) given flow state \( (y = [x, p]) \) where \( x \) is state and \( p \) is costate

**Parameters**
- \( \mathbf{y} \ (\text{array_like, shape} \ (2N,)) \) – Flow state, \( y = [x, p] \), where \( x \) is state and \( p \) is costate
- \( t \ (\text{float}) \) – Time value

**Returns** \( \mathbf{pd} \) – Costate dynamics given current flow state and time

**Return type** array_like, shape \((N,)\)

\( \text{dPhidt}(\mathbf{Y}, t) \)
Computes state transition matrix STM dynamics \( \frac{d\Phi}{dt} \) given flow state \( (Y = [x, p, \Phi]) \) where \( x \) is state, \( p \) is costate, and \( \Phi \) is STM for \( y \)

**Parameters**
- \( \mathbf{Y} \ (\text{array_like, shape} \ (2N + (2N)^2,)) \) – Augmented flow state, \( \mathbf{Y} = [x, p, \text{vec(Phi)}] \), where \( x \) is state, \( p \) is costate, and \( \Phi \) is STM for \( y \)
- \( t \ (\text{float}) \) – Time value

**Returns** \( \mathbf{Phid} \) – State dynamics given current flow state and time

**Return type** array_like, shape \((2N,2N)\)

\( \mathbf{H}(\mathbf{y}, t) \)
Computes optimal control Hamiltonian given flow state \( (y = [x, p]) \) where \( x \) is state and \( p \) is costate

**Parameters**
- \( \mathbf{y} \ (\text{array_like, shape} \ (2N,)) \) – Flow state, \( y = [x, p] \), where \( x \) is state and \( p \) is costate
- \( t \ (\text{float}) \) – Time value

**Returns** \( \mathbf{HVal} \) – Optimal control Hamiltonian value given current flow state and time

**Return type** float

\( \text{propagate}_\mathbf{y}(\mathbf{y}_0, t_0, t_f, \text{numTimeSteps=None, rtol=1e-06, atol=1e-06}) \)
Propagate flow state \( \mathbf{y}_0 = [x_0, p_0] \) from \( t_0 \) to \( \mathbf{y}_f = [x_f, p_f] \) at \( t_f \)
Parameters

- \( \mathbf{y}_0 \) (array_like, shape \((2N,))\) – Initial flow state, \( \mathbf{y}_0 = [\mathbf{x}_0, \mathbf{p}_0] \), where \( \mathbf{x}_0 \) is initial state and \( \mathbf{p}_0 \) is initial costate
- \( t_0 \) (float) – Initial time value
- \( t_f \) (float) – Final time value
- \( \text{numTimeSteps} \) (int, optional) – If not provided, this function will only return \( \mathbf{y}_f = [\mathbf{x}_f, \mathbf{p}_f] \)
  If provided, this function will return \( \mathbf{x}(t), \mathbf{p}(t) \) with \( t \) from \( t_0 \) to \( t_f \) with \( \text{numTimeSteps} \) time values
- \( \text{atol} \) (rtol,) – Relative and absolute tolerance for the differential equation solver

Returns \( \mathbf{y}_f \) – Depending on the input argument of \( \text{numTimeSteps} \), this will either be the final flow state \( \mathbf{y}_f = [\mathbf{x}_f, \mathbf{p}_f] \) or it would be an array with \( \mathbf{y}(t) \) from \( t_0 \) to \( t_f \) with \( \text{numTimeSteps} \) time values

Return type array_like, shape \((2N,)\) or \((\text{numTimeSteps},2N)\)

.. function:: propagate_Y(\mathbf{y}_0, t_0, t_f, \text{numTimeSteps}=None, rtol=1e-06, atol=1e-06)

Propagate augmented flow state \( \mathbf{Y}_0 = [\mathbf{x}_0, \mathbf{p}_0, \text{vec(\Phi}_0)] \) from \( t_0 \) to \( \mathbf{Y}_f = [\mathbf{x}_f, \mathbf{p}_f, \text{vec(\Phi}_f)] \) at \( t_f \)

Parameters

- \( \mathbf{y}_0 \) (array_like, shape \((2N,))\) – Initial flow state, \( \mathbf{y}_0 = [\mathbf{x}_0, \mathbf{p}_0] \), where \( \mathbf{x}_0 \) is initial state and \( \mathbf{p}_0 \) is initial costate
- \( t_0 \) (float) – Initial time value
- \( t_f \) (float) – Final time value
- \( \text{numTimeSteps} \) (int, optional) – If not provided, this function will only return \( \mathbf{Y}_f = [\mathbf{x}_f, \mathbf{p}_f, \text{vec(\Phi}_f)] \)
  If provided, this function will return \( \mathbf{x}(t), \mathbf{p}(t) \) with \( t \) from \( t_0 \) to \( t_f \) with \( \text{numTimeSteps} \) time values
- \( \text{atol} \) (rtol,) – Relative and absolute tolerance for the differential equation solver

Returns \( \mathbf{Y}_f \) – Depending on the input argument of \( \text{numTimeSteps} \), this will either be the final augmented flow state \( \mathbf{Y}_f = [\mathbf{x}_f, \mathbf{p}_f, \text{vec(\Phi}_f)] \) or it would be an array with \( \mathbf{Y}(t) \) from \( t_0 \) to \( t_f \) with \( \text{numTimeSteps} \) time values

Return type array_like, shape \((2N + (2N)^2,\) or \((\text{numTimeSteps},2N + (2N)^2)\)

class dynSystems.dynSystemTemplate

Bases: object

Template for dynamic system class

__init__()

Constructor for Dynamic System Class

dydt(\mathbf{y}, t)

Computes trajectory flow dynamics \( (\text{dy/dt}) \) given flow state \( \mathbf{y} = [\mathbf{x}, \mathbf{p}] \) where \( \mathbf{x} \) is state and \( \mathbf{p} \) is costate

Parameters

- \( \mathbf{y} \) (array_like, shape \((2N,))\) – Flow state, \( \mathbf{y} = [\mathbf{x}, \mathbf{p}] \), where \( \mathbf{x} \) is state and \( \mathbf{p} \) is costate
- \( t \) (float) – Time value
**dYdt**(*Y*, *t*)
Computes augmented flow dynamics (dPhi/dt) given flow state (*Y* = [x, p, Phi]) where x is state, p is costate, and Phi is STM for y

**Parameters**

- **Y** *(array_like, shape (2N + (2N)^2,)) –* Augmented flow state, *Y* = [x, p, vec(Phi)], where x is state, p is costate, and Phi is STM for y
- **t** *(float) –* Time value

**Returns** *Yd* – Augmented flow dynamics given current flow state and time

**Return type** array_like, shape (2N + (2N)^2,)

**dxdt**(*y*, *t*)
Computes state dynamics (dx/dt) given flow state (*y* = [x, p]) where x is state and p is costate

**Parameters**

- **y** *(array_like, shape (2N,)) –* Flow state, *y* = [x, p], where x is state and p is costate
- **t** *(float) –* Time value

**Returns** *xd* – State dynamics given current flow state and time

**Return type** array_like, shape (N,)

**dpdt**(*y*, *t*)
Computes costate dynamics (dp/dt) given flow state (*y* = [x, p]) where x is state and p is costate

**Parameters**

- **y** *(array_like, shape (2N,)) –* Flow state, *y* = [x, p], where x is state and p is costate
- **t** *(float) –* Time value

**Returns** *pd* – Costate dynamics given current flow state and time

**Return type** array_like, shape (N,)

**dPhidt**(*Y*, *t*)
Computes state transition matrix STM dynamics (dPhi/dt) given flow state (*Y* = [x, p, Phi]) where x is state, p is costate, and Phi is STM for y

**Parameters**

- **Y** *(array_like, shape (2N + (2N)^2,)) –* Augmented flow state, *Y* = [x, p, vec(Phi)], where x is state, p is costate, and Phi is STM for y
- **t** *(float) –* Time value

**Returns** *Phid* – State dynamics given current flow state and time

**Return type** array_like, shape (2N,2N)

**H**(*y*, *t*)
Computes optimal control Hamiltonian given flow state (*y* = [x, p]) where x is state and p is costate

**Parameters**
• \(y\) (array_like, shape \((2N,)\)) – Flow state, \(y = \{x, p\}\), where \(x\) is state and \(p\) is costate

• \(t\) (float) – Time value

Returns HVal – Optimal control Hamiltonian value given current flow state and time

Return type float

propagate_y \((y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)\)
Propagate flow state \(y0 = [x0, p0]\) from \(t0\) to \(yf = [xf, pf]\) at \(tf\)

Parameters

• \(y0\) (array_like, shape \((2N,)\)) – Initial flow state, \(y0 = \{x0, p0\}\), where \(x0\) is initial state and \(p0\) is initial costate

• \(t0\) (float) – Initial time value

• \(tf\) (float) – Final time value

• numTimeSteps (int, optional) – If not provided, this function will only return \(yf = [xf, pf]\)

If provided, this function will return \(x(t), p(t)\) with \(t\) from \(t0\) to \(tf\) with numTimeSteps time values

• atol (rtol,) – Relative and absolute tolerance for the differential equation solver

Returns yf – Depending on the input argument of numTimeSteps, this will either be the final flow state \(yf = [xf, pf]\) or it would be an array with \(y(t)\) from \(t0\) to \(tf\) with numTimeSteps time values

Return type array_like, shape \((2N,)\) or \((numTimeSteps,2N)\)

propagate_Y \((y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06)\)
Propagate augmented flow state \(Y0 = \{x0, p0, vec(\Phi0)\}\) from \(t0\) to \(Yf = \{xf, pf, vec(\Phi f)\}\) at \(tf\)

Parameters

• \(y0\) (array_like, shape \((2N,)\)) – Initial flow state, \(y0 = \{x0, p0\}\), where \(x0\) is initial state and \(p0\) is initial costate

• \(t0\) (float) – Initial time value

• \(tf\) (float) – Final time value

• numTimeSteps (int, optional) – If not provided, this function will only return \(Yf = [xf, pf, vec(\Phi f)]\)

If provided, this function will return \(x(t), p(t)\) with \(t\) from \(t0\) to \(tf\) with numTimeSteps time values

• atol (rtol,) – Relative and absolute tolerance for the differential equation solver

Returns Yf – Depending on the input argument of numTimeSteps, this will either be the final augmented flow state \(Yf = \{xf, pf, vec(\Phi f)\}\) or it would be an array with \(Y(t)\) from \(t0\) to \(tf\) with numTimeSteps time values

Return type array_like, shape \((2N + (2N)^2,)\) or \((numTimeSteps,2N + (2N)^2)\)
8.5 qLPV System Module

class qLPV.qLPV(A, B, v, xtrm, utrm, vFunc, dv_dxFunc, dv2_dx2Func, controlConstraint, etrm=None, hFunc=None, dh_dxtFunc=None, dh2_dxt2Func=None, dh_detFunc=None, dh2_det2Func=None, HFunc=None, dH_dxFunc=None, dH2_dx2Func=None)

Bases: object

Class for qLPVs and evaluation of qLPV derivatives xdot = A(v)*(x - x_v(v)) + B(v)(u - u_v(v)) - h( e_v(v), x_v(v)) + H(x)

where v(x)

__init__(A, B, v, xtrm, utrm, vFunc, dv_dxFunc, dv2_dx2Func, controlConstraint, etrm=None, hFunc=None, dh_dxtFunc=None, dh2_dxt2Func=None, dh2_det2Func=None, HFunc=None, dH_dxFunc=None, dH2_dx2Func=None)

Dimensions:

n = dimension of state (x), costate (p)
m = dimension of control (u)
a = dimension of trim parameters
b = number of tabulated values for each of the given parameters

A : n x n x b : State dynamics matrix
B : n x m x b : Control input matrix
v : 1 x b or (b,) : scalar qLPV parameter
xtrm : n x b : Trim states
utrm : m x b : Trim control
etrm : a x b : Trim parameters, optional

vFunc : callable function that takes state, x as the input and outputs the scalar v
dv_dxFunc : callable function that takes state, x as the input and outputs the jacobian dv/dx (1 x n or n,)
dv2_dx2Func : callable function that takes state, x as the input and outputs the hessian dv2/dx2 (n x n)

controlConstraint : constraint object representing constraints on control inputs

hFunc : callable function that takes xtrim and returns trim function (n, or n x 1), optional, if not given, assumed zero
dh_dxtFunc : callable function that takes xtrim and returns trim function jacobian with respect to trim state (n x n), optional, if not given, assumed zero
dh2_dxt2Func : callable function that takes xtrim and returns trim function double jacobian with respect to trim state (n x n x n), optional, if not given, assumed zero
dh_detFunc : callable function that takes etrim and returns trim function jacobian with respect to trim parameters (n x a), optional, if not given, assumed zero
dh2_det2Func : callable function that takes etrim and returns trim function double jacobian with respect to trim parameters (n x a x a), optional, if not given, assumed zero
HFunc : callable function that takes x and returns nonlinear portion of dynamics (n, or n x 1), optional, if not given, assumed zero

dH_dxFunc : callable function that takes x and returns nonlinear dynamics jacobian with respect to state (n x n), optional, if not given, assumed zero

dH2_d2xFunc : callable function that takes x and returns nonlinear dynamics double jacobian with respect to state (n x n x n), optional, if not given, assumed zero

A ()
Updates qLPV A arrays by evaluating spline fit at given state

B ()
Updates qLPV B arrays by evaluating spline fit at given state

xtrm ()
Updates qLPV xtrm array by evaluating spline fit at given state

utrm ()
Updates qLPV utrm array by evaluating spline fit at given state

etrm ()
Updates qLPV etrm array by evaluating spline fit at given state

uStar ()
Computes optimal control

optimalControlJacobian ()
Computes jacobians of optimal control input with respect to state and costate

update_qLPV (x, p, withJacobian=False)
Updates parameter and all other states based off of state and costate input

update_jacobian (x, p)
Update jacobian terms based on state and costate input

trimUpdate ()
Given current state, x, update trim function (h), and its derivatives

nonlinearTermUpdate ()
Given current state, x, update nonlinear dynamics section of dynamics value

dfdx ()
Updates dfdx given current state

dfdp ()
Updates dfdp given current state

dgdp ()
Updates dgdp given current state

dgdx ()
Updates dgdx given current state

dydt (y, t)
Flow dynamics

dxdt (y, t)
State Dynamics

dpdt (y, t)
Costate dynamics

8.5. qLPV System Module
\begin{align*}
\text{dPhidt}(Y, t) & \quad \text{STM Dynamics} \\
\text{dYdt}(Y, t) & \quad \text{Augmented Flow dynamics} \\
\text{propagate}_y(y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06) & \quad \text{Propagate } y_0 = x_0, p_0 \text{ from } t_0 \text{ to } y_f = x_f, p_f \text{ at } t_f \\
\text{propagate}_Y(Y0, t0, tf, numTimeSteps=None, rtol=1e-06, atol=1e-06) & \quad \text{Propagate } Y_0 = x_0, p_0, \Phi_0 \text{ from } t_0 \text{ to } y_f = x_f, p_f, \Phi_f \text{ at } t_f \\
\text{plotqLPVFit}(new_v=None, showAxes=False) & \quad \text{Plots qLPV spline fit with all of the table lookup quantities}
\end{align*}

### 8.6 Continuation Module

\texttt{contMethodMod.contMethod}_T(Tvec, particle, dynamics, contMethodOption=2, newtonsCorrection=False)

This function computes continuation methods over time horizon, \( T \), for the given particle. It updates the fields of the given particle directly.

**Parameters**

- \( Tvec \) (array_like, shape \((\text{numTimeHorizonSteps},)\)) – Array of time horizon, \( T \), values to evaluate the continuation method over.
- \( \text{particle} \) (Particle object) – Current particle in reachability analysis. This is parametrized by a hyperspherical coordinate or unit vector search direction.
- \( \text{dynamics} \) (Dynamics object) – Dynamics system object that can propagate the state and costate to the specified time horizon.
- \( \text{contMethodOption} \) (int, optional) – Integer specifying which homotopy/continuation method is desired.
  - Options are:
    1. Basic continuation method
    2. Pseudo-arclength continuation method
    3. Pseudo-arclength homotopy using Sard’s theorem (NOT WORKING)
    4. Predictor-Corrector with Newton step updates (IN DEVELOPMENT)
    5. Predictor-Corrector with Broyden step updates (IN DEVELOPMENT)
- \( \text{newtonsCorrection} \) (bool, optional) – If true, a Newton’s method correction will be performed after the continuation method is complete.

\texttt{contMethodMod.contMethodODE}(z, T, particle, dynamics)

This is the ODE function for the basic continuation method integrator routines over time horizon. It updates the fields of the given particle directly.

**Parameters**

- \( z \) (array_like, shape \((N+1,)\)) – Concatenation between initial state and Lagrange multiplier for the initial condition constraint, \( z = [x_0, \lambda] \)
• **T** (<code>float</code>) – Current time horizon in the continuation method integration

• **particle** (<code>Particle object</code>) – Current particle in reachability analysis. This is parametrized by a hyperspherical coordinate or unit vector search direction

• **dynamics** (<code>Dynamics object</code>) – Dynamics system object that can propagate the state and costate to the specified time horizon

```python
contMethodMod.arcLengthEvent1(z_s, sig, particle)
contMethodMod.arcLengthEvent1s(z_s, sig, particle)
contMethodMod.arcLengthEvent2(z_s, sig, particle)
contMethodMod.arcLengthEvent3(z_s, sig, particle)
contMethodMod.arcLengthEvent4(z_s, sig, particle)

contMethodMod.contMethodODE_arcLength(z_s, sig, particle, dynamics)
```

This is the ODE function for the pseudoarclength continuation method integrator routines over time horizon. It updates the fields of the given particle directly.

**Parameters**

• **z_s** (<code>array_like, shape (N+2,)</code>) – Concatenation between initial state, Lagrange multiplier for the initial condition constraint, and current time horizon in continuation method integration

  <i>z = [x0, lam, T]</i>

• **sig** (<code>float</code>) – Current arc length in the pseudo arclength continuation method integration

• **particle** (<code>Particle object</code>) – Current particle in reachability analysis. This is parametrized by a hyperspherical coordinate or unit vector search direction

• **dynamics** (<code>Dynamics object</code>) – Dynamics system object that can propagate the state and costate to the specified time horizon

```python
contMethodMod.contMethodODE_arcLength_generalized(z_tau, sig, particle, dynamics, case=1)
contMethodMod.contMethodODE_arcLength_Sard(z_s, sig, particle, dynamics)
contMethodMod.newtonsCorrection_dFdz(particle, dynamics, convTol=1e-06, maxIter=5)
contMethodMod.computeTangentVector(dFdz, dFdT, tol=1e-09)
contMethodMod.PC_Continuation(particle, dynamics, h, hmin, tol, dmax, ctmax, hmax)
contMethodMod.PC_Continuation2(particle, dynamics, h, hmin, hmax, tol, maxNewtIter=3)
contMethodMod.PC_Continuation_Broyden(particle, dynamics, h, hmin, tol)
contMethodMod.PC_Continuation_Broyden2(particle, dynamics, h, hmin, hmax, tol)
contMethodMod.contMethod_toT(Tdesired, particle, dynamics, newtonsCorrection=False)
contMethodMod.contMethod(Tvec, particle, dynamics, newtonsCorrection=False)
```

### 8.7 Plotting Module

```python
plotting.axisEqual(ax, axesStr='xyz')
```

3D Equivalent of axis equal for matplotlib

---

8.7. Plotting Module 53
Parameters

- `ax (axis object) – Matplotlib axis object`
- `axesStr (str, optional) – String of axes (e.g. ‘xy’, ‘xyz’, ‘xz’) that list axes that should have equal distance units`

`plotting.plotReachTrajectories (particles, projDim=None, sc=1.0, axesLabels=None, addedPtInds=None, axisEqualBool=True)`

Plot particle state trajectories in state/phase space at current time horizon $T$ ($x(t;T)$). This also highlights the initial states ($x_0(T) = x(t_0;T)$) and final states ($x_f(T) = x(t_f;T)$) with markers. Note that the only components of the state that are plotted will be the ones specified by the input argument `projDim`.

This function has the capability of plotting different components of the state vector using the `projDim` argument.

This function can only plot 2D or 3D state trajectories.

This function only pulls from the `particle.x_t` attribute to create the plot.

Parameters

- `particles (list of Particle objects) – List of particle objects created for a reachability analysis that need to be plotted. This function plots details from every particle in this argument list`
- `projDim (list of int, optional) – Projection dimensions are a list of which dimensions of the state vector to plot trajectories of (e.g. `projDim` = [1,3] will plot the first and third components of the state vector)`

NOTICE: This uses MATLAB-based indexing where `projDim` = [1,2] will plot the first and second state.

By default, if no value is specified, `projDim` = [1,2] and function will plot the first and second state trajectories in state/phase space.

The length of `projDim` should be either 2 or 3.

- `sc ({float, list of floats}, optional) – All plotted quantities are multiplied by `sc` before plotting. This is useful in cases where units of the reachability analysis are different from the desired units of the plot. If this is a list, it should correspond to a scaling factor for each dimension/axis being plotted`
- `axesLabels (list of str, optional) – If not provided, the default axis labels are based on the `projDim` argument (e.g. ‘x1’, ‘x2’). If this argument is provided, each entry in this list will be the axis label for the axes specified in `projDim` This argument, if provided, should be the same length as the `projDim` argument.
- `addedPtInds (list of int, optional) – This argument is a list of indices of the `particles` argument list that correspond to added points through mesh refinement or some other process. Every particle with an index specified by this argument will have a different marker than the original particles`

Returns `fh` – A figure handle for the created plot.

Return type Figure handle

`plotting.plotReachHistories1D (particles, projDim=1, shadedTrajBool=True, particleHistBool=False, sc=1.0, axesLabels=None)`

Plot particle final states over time horizon $T$ for a single dimension of the state space (i.e. $x_{f_i}(T) = x_i(t_f;T)$ where $i$ denotes the specified dimension of the state to plot)
This is equivalent to projecting the reachable volume onto the specified state dimension. This way, this plot shows the reachable envelope (range of all possible state values) for the specified state dimension as a function of time horizon $T$

This function is the 1-dimensional version of `plotReachGrowth`

The `projDim` argument specifies which state dimension to plot. This must be a single value

This function pulls from the `particle.xf_T` and `particle.xf_T_Spline` attributes to create the plot

**Parameters**

- `particles` *(list of Particle objects)* – List of particle objects created for a reachability analysis that need to be plotted. This function plots details from every particle in this argument list
- `projDim` *(int, optional)* – Projection dimension is an integer specifying which dimensions of the state vector to plot details of (e.g. `projDim = 1` will plot the first component of the state vector).

NOTICE: This uses MATLAB-based indexing where `projDim = 1` will plot the first state information
- `shadedTrajBool` *(bool, optional)* – This boolean determines whether or not the set of possible values for the specified state is shaded gray or not.
- `particleHistBool` *(bool, optional)* – This boolean determines whether or not the individual particle final state histories histories ($xf_i(T) = x_i(tf;T)$ where i specifies the dimension of the state that is being plotted)

If True, the individual particle histories will be drawn as dashed lines.
- `sc` *(float, optional)* – All plotted quantities are multiplied by `sc` before plotting. This is useful in cases where units of the reachability analysis are different from the desired units of the plot.
- `axesLabels` *(list of str, optional)* – If not provided, the default axis labels are based on the `projDim` argument (e.g. ‘x1’, ‘x2’) and the horizontal axis label defaults to ‘T’ for time horizon.

If this argument is provided, each entry in this list will be the axis label for the axes specified in `projDim`

This argument, if provided, should be the same length as the `projDim` argument.

**Returns**

- `fh` – A figure handle for the created plot

**Return type** Figure handle

```python
plotting.plotReachGrowth(particles, numSnaps=5, projDim=None, sc=1.0, axesLabels=None, axisEqualBool=True)
```

Plots reachable volume at different snapshots of time horizon, $T$ (from $T=0$ to the final time horizon). This plot shows how the reachable volume is evolving over time horizon

This plot represents the reachable volume as a polygon in this plot by connecting a straight line between each particle state

This function is the multi-dimensional version of `plotReachHistories1D`

The `projDim` argument specifies which state dimensions to plot.

This function pulls from the `particle.xf_T` and `particle.xf_T_Spline` attributes to create the plot

**Parameters**

8.7. Plotting Module 55
• **particles** *(list of Particle objects)* – List of particle objects created for a reachability analysis that need to be plotted. This function plots details from every particle in this argument list.

• **numSnaps** *(int)* – Specifies number of reachable volumes to plot from T=0 to the final time horizon (including the both endpoints).

In the case where `numSnaps = 1`, the final time horizon, only the final time horizon reachable volume will be plotted.

The time horizon, T, values will be equally spaced from 0 to T_f including both endpoints.

• **projDim** *(list of int, length 2 or 3, optional)* – Projection dimensions are a list of which dimensions of the state vector to plot trajectories of (e.g. `projDim = [1,3]` will plot the first and third components of the state vector).

NOTICE: This uses MATLAB-based indexing where `projDim = [1,2]` will plot the first and second state.

By default, if no value is specified, `projDim = [1,2]` and function will plot the first and second state trajectories in state/phase space.

• **sc** *(float, list of floats), optional)* – All plotted quantities are multiplied by `sc` before plotting. This is useful in cases where units of the reachability analysis are different from the desired units of the plot.

If this is a list, it should correspond to a scaling factor for each dimension/axis being plotted.

• **axesLabels** *(list of str, optional)* – If not provided, the default axis labels are based on the `projDim` argument (e.g. ‘x1’, ‘x2’).

If this argument is provided, each entry in this list will be the axis label for the axes specified in `projDim`.

This argument, if provided, should be the same length as the `projDim` argument.

**Returns** *fh* – A figure handle for the created plot

**Return type** Figure handle

```python
plotting.plotReachVolume(particles, projDim=None, sc=1.0, axesLabels=None, axisEqualBool=True)
```

Plots reachable volume at current/final time horizon, T. This plot shows the current reachable volume at the current time horizon, T, with markers representing the particle final states and lines denoting neighbor relationships between particles.

The `projDim` argument specifies which state dimensions to plot.

This function only pulls from the `particle.xf_T` attribute to create the plot.

**Parameters**

• **particles** *(list of Particle objects)* – List of particle objects created for a reachability analysis that need to be plotted. This function plots details from every particle in this argument list.

• **projDim** *(list of int, length 2 or 3, optional)* – Projection dimensions are a list of which dimensions of the state vector to plot trajectories of (e.g. `projDim = [1,3]` will plot the first and third components of the state vector).

NOTICE: This uses MATLAB-based indexing where `projDim = [1,2]` will plot the first and second state.

By default, if no value is specified, `projDim = [1,2]` and function will plot the first and second state trajectories in state/phase space.
• \texttt{sc} (\texttt{\{float, list of floats\}, optional}) – All plotted quantities are multiplied by \texttt{sc} before plotting. This is useful in cases where units of the reachability analysis are different from the desired units of the plot.

If this is a list, it should correspond to a scaling factor for each dimension/axis being plotted

• \texttt{axesLabels} (\texttt{list of str, optional}) – If not provided, the default axis labels are based on the \texttt{projDim} argument (e.g. ‘x1’, ‘x2’).

If this argument is provided, each entry in this list will be the axis label for the axes specified in \texttt{projDim}

This argument, if provided, should be the same length as the \texttt{projDim} argument.

\textbf{Returns} \texttt{fh} – A figure handle for the created plot

\textbf{Return type} Figure handle

\begin{verbatim}
plotting.plotIntersection(R1, R2, R1color=None, R2color=None, sc=1.0, axesLabels=None, axisEqualBool=True)
\end{verbatim}

Takes in two reachability objects, computes intersection of them in the subspace of interest, and plots result. It also returns a list of particles that make up the intersection (if they exist)

This function only pulls from the \texttt{particle.xf_T} attribute to create the plot

\textbf{Parameters}

• \texttt{R1} (\texttt{Reachability object}) – First Reachability object to use in intersection computation

• \texttt{R2} (\texttt{Reachability object}) – Second Reachability object to use in intersection computation

• \texttt{R1color} (\texttt{rgb color list, optional}) – RGB Color list that will denote the color used to plot R1

• \texttt{R2color} (\texttt{rgb color list, optional}) – RGB Color list that will denote the color used to plot R2

• \texttt{sc} (\texttt{\{float, list of floats\}, optional}) – All plotted quantities are multiplied by \texttt{sc} before plotting. This is useful in cases where units of the reachability analysis are different from the desired units of the plot.

If this is a list, it should correspond to a scaling factor for each dimension/axis being plotted

• \texttt{axesLabels} (\texttt{list of str, optional}) – If not provided, the default axis labels are based on the \texttt{projDim} argument (e.g. ‘x1’, ‘x2’).

If this argument is provided, each entry in this list will be the axis label for the axes specified in \texttt{projDim}

This argument, if provided, should be the same length as the \texttt{projDim} argument.

\textbf{Returns}

• \texttt{fh} (\texttt{Figure handle}) – A figure handle for the created plot

• \texttt{intersectParticles} (\texttt{list of Particle objects}) – A list of particles from R1 and/or R2 that are intersection particles. That is, they belong to both reachable volumes simultaneously. If the two reachable volumes do not intersect, this list will be empty.

\begin{verbatim}
plotting.plotUnion(R1, R2, R1color=None, R2color=None, sc=1.0, axesLabels=None, axisEqualBool=True)
\end{verbatim}

Takes in two reachability objects, computes union of them in the subspace of interest, and plots result. It also returns a list of particles that make up the union

\section{Plotting Module}
This function only pulls from the `particle.xf_T` attribute to create the plot

**Parameters**

- **R1** *(Reachability object)* – First Reachability object to use in union computation
- **R2** *(Reachability object)* – Second Reachability object to use in union computation
- **R1color** *(rgb color list, optional)* – RGB Color list that will denote the color used to plot R1
- **R2color** *(rgb color list, optional)* – RGB Color list that will denote the color used to plot R2
- **sc** *(float, list of floats), optional* – All plotted quantities are multiplied by `sc` before plotting. This is useful in cases where units of the reachability analysis are different from the desired units of the plot.

  If this is a list, it should correspond to a scaling factor for each dimension/axis being plotted

- **axesLabels** *(list of str, optional)* – If not provided, the default axis labels are based on the `projDim` argument (e.g. ‘x1’, ‘x2’).

  If this argument is provided, each entry in this list will be the axis label for the axes specified in `projDim`

  This argument, if provided, should be the same length as the `projDim` argument.

**Returns**

- **fh** *(Figure handle)* – A figure handle for the created plot
- **unionParticles** *(list of Particle objects)* – A list of particles from R1 and/or R2 that are union particles. That is, they belong to only one of the reachable volumes boundary. If the two reachable volumes do not intersect, to returning all the particles.

plotting.plotRedistributionCDFs(meshRefinementBeforeAfter, idealPlotBool=True, columnBool=True)

Plot cumulative distribution function (cdf) type plots for the particle redistribution problem

This results in 3 plots being created

1. A cdf plot based on distribution of all edge-wise (particle to particle pair) distances/costs (over all edges)
2. A cdf plot based on distribution of all particle-wise distances/costs (over all particles)
3. A line plot showing change in overall cost before and after each mesh refinement iteration for all iterations

**Parameters**

- **meshRefinementBeforeAfter** *(list)* – List of mesh refinements properties both before and after refinement iterations.

  Each index (corresponds to mesh refinement iteration) has [before, after] were before/after are J, JHist, JHist_edge (or D equivalents)

- **idealPlotBool** *(bool, optional)* – If True, then include the “ideal” cdf graph to help visualize what the ideal particle distribution looks like. This should be a step-like function at the average distance of the particle final states

- **columnBool** *(bool, optional)* – If True, then single columns of subplots will be made where each subplot corresponds to a mesh refinement iteration.

  If False, then all cdfs are shown on a single plot

**Returns**
• **redistCdfEdgeFig** (*Figure handle*) – A figure handle corresponding to the cdf plot based on edge distances

• **redistCdfParticleFig** (*Figure handle*) – A figure handle corresponding to the cdf plot based on particle-wise distances/cost

• **JSaveFig** (*Figure handle*) – A figure handle corresponding to the line plot of the overall cost change over mesh refinement iterations

### 8.8 Utilities Module

This module contains useful functions.

**utilities.R1** (*th*, *deg=None*)

Performs a coordinate/frame transformation about the x-axis

**Parameters**

- **th** (*float*) – Angle in radians to rotate (default radians)
- **deg** (*bool*, optional) – If set, the input will be treated as degrees

**Returns**  
- **out** – 3x3 Coordinate/Frame transformation matrix  
- **Return type**  
  array_like, shape(3,3)

**utilities.R2** (*th*, *deg=None*)

Performs a coordinate/frame transformation about the y-axis

**Parameters**

- **th** (*float*) – Angle in radians to rotate (default radians)
- **deg** (*bool*, optional) – If set, the input will be treated as degrees

**Returns**  
- **out** – 3x3 Coordinate/Frame transformation matrix  
- **Return type**  
  array_like, shape(3,3)

**utilities.R3** (*th*, *deg=None*)

Performs a coordinate/frame transformation about the z-axis

**Parameters**

- **th** (*float*) – Angle in radians to rotate (default radians)
- **deg** (*bool*, optional) – If set, the input will be treated as degrees

**Returns**  
- **out** – 3x3 Coordinate/Frame transformation matrix  
- **Return type**  
  array_like, shape(3,3)

**utilities.matNorm** (*x*)

return the norm of every row of a numpy array

**utilities.normOfVec** (*x*)

Faster norm of 1D vector that np.linalg.norm for small arrays

**utilities.normalize** (*vec*)

normalize input array

**utilities.angBtwnVecs** (*u, v*)

Return angle (radians) between two input vectors
utilities.angleBtwnUnitVecs(u, v)
    Return angle (radians) between two input unit vectors

utilities.adj(matrix)
    Adjoint of matrix

utilities.getOrthProjection(ofv1, ontov2)
    Get the orthogonal projection of v1 onto v2. Returns orthogonal projection of v1 onto v2 and projection of v1 onto v2

utilities.vec2radec(vec)
    Converts vec to ra/dec [deg]

utilities.vec2radec_rad(vec)
    Converts vec to ra/dec [rad]

utilities.radec2vec(ra, dec)
    Converts ra/dec [deg] to vector

utilities.radec2vec_rad(ra, dec)
    Converts ra/dec [rad] to vector

utilities.randInRange(N, minVal, maxVal)
    Returns random array of shape (N,) from minVal to maxVal

utilities.numericalJacobian(f, y, h=0.0001, jacType='central')
    Computes the numerical jacobian of f(y), dfdy If f returns a n x m array and y is a l x 1 or (l,) array, then output jacobian is squeeze(n x m x l) If f returns a (n,) array and y is a l x 1 (l,) array, then output jacobian is squeeze(n x l)

    Parameters
    • f (callable function) – This is the function that must be callable like f(y)
    • y (array_like, shape (l,) or (l,1)) – This is a single vector/array that the numerical jacobian will be computed at
    • h (float, optional) – Step size
    • jacType (central, forward, backward, optional) – Type of numerical differencing to use

    Returns dfdy – Numerical jacobian of function f around point y (dfdy(y))

    Return type array_like

utilities.sympyNorm(x)
    Takes in a sympy vector as a Matrix object and returns a sympy object that represents the 2 norm

    Parameters x (sympy Matrix object) – sympy vector to take the 2 norm of

    Returns s – Symbolic expression of 2 norm of input vector x

    Return type sympy expression

utilities.sympySignApprox(x, epsil=1e-06, scalar=False)
    Takes in a sympy vector as a Matrix object and returns a sympy object that represents a smooth approximation of the sign/signum function.

    Parameters
    • x (sympy Matrix object) – sympy vector to take the sign of
    • epsil (float, optional) – Small number for the sign approximation. As epsil approaches zero, the approximation approaches the true sign function
• `scalar (bool, optional)` – Boolean that says whether or not the input \( x \) is a scalar (not a Matrix object)

Returns `sign_x` – Symbolic expression sign of vector \( x \)

Return type sympy expression

utilities.\texttt{sympyAbsApprox}(x, \texttt{epsil}=1e-06, scalar=False)

Takes in a sympy vector as a Matrix object and returns a sympy object that represents a smooth approximation of the absolute value function.

Parameters

• \( x \) (sympy Matrix object) – sympy vector to take the absolute value of

• `epsil (float, optional)` – Small number for the absolute value approximation. As \( epsil \) approaches zero, the approximation approaches the true absolute value function

• `scalar (bool, optional)` – Boolean that says whether or not the input \( x \) is a scalar (not a Matrix object)

Returns `abs_x` – Symbolic expression absolute value of vector \( x \)

Return type sympy expression

utilities.\texttt{sympySatApprox}(x, \texttt{epsil}=1e-06, scalar=False)

Takes in a sympy vector as a Matrix object and returns a sympy object that represents a smooth approximation of the unit saturation function. The linear region of inputs must be from -1 to 1 and the resulting output will be from -1 to 1.

Parameters

• \( x \) (sympy Matrix object) – sympy vector to take the absolute value of

• `epsil (float, optional)` – Small number for the absolute value approximation. As \( epsil \) approaches zero, the approximation approaches the true absolute value function

• `scalar (bool, optional)` – Boolean that says whether or not the input \( x \) is a scalar (not a Matrix object)

Returns `sat_x` – Symbolic expression absolute value of vector \( x \)

Return type sympy expression

utilities.\texttt{printProgressBar}(iteration, total, prefix="", suffix="", decimals=1, length=100, fill='[U+2588]'

Print progress bar in terminal. Should call in a loop to create terminal progress bar.

Parameters

• `iteration (int)` – Current iteration

• `total (int)` – Total iterations

• `prefix (str, optional)` – Prefix string

• `suffix (str, optional)` – Suffix string

• `decimals (int, optional)` – Positive number of decimals in percent complete

• `length (int, optional)` – Character length of bar

• `fill (str, optional)` – Bar fill character

utilities.\texttt{null}(a, \texttt{rtol}=2.220446049250313e-16)

Compute null space of matrix using svd

8.8. Utilities Module

61
utilities.qr_null(A, tol=None)
    Compute null space of matrix using QR

utilities.QR_null(A)
    Compute null space of matrix using QR and only returns a single dimension of the null space. Assumes matrix
    A is N x N+1 with rank N

utilities.outliers_z_score(ys, threshold=3)
    detect outlier indices in 1D data using z-score (1D Mahalanobis dist)

utilities.outliers_modified_z_score(ys, threshold=3.5)
    detect outlier indices in 1D data using modified z-score which is more robust

utilities.outliers_iqr(ys, q1=25, q3=75, upperOnly=True)
    Detect outlier indices in 1D data using IQR method (pretty robust)

Parameters
    • ys (array_like) – Numpy array of data values to compute outliers of
    • q1 (float, optional) – Lower quartile value
    • q3 (float, optional) – Upper quartile value
    • upperOnly (bool, optional) – If True, only returns outliers on the positive side of
      the distribution (larger than q3)

utilities.cpickleSave(saveList, filename='obj.save')
    Saves variables in the saveList as a pickle file

utilities.cpickleLoad(filename='obj.save')
    Loads variables stored in pickle file as list

utilities.dillSave(saveList, filename='obj.save')
    Uses dill to save variables in the saveList

utilities.dillLoad(filename='obj.save')
    Loads variables stored by dill as list

utilities.createMultiDimList(m, n)
    Creates a m x n list array with all 0s
    Index like sampleList[i][j]

utilities.removeEntriesFromDict(entries, the_dict)
    Remove entries in entries from the_dict

utilities.generateThetaArray(numThDim, uniformSamplingRate, uniformlySpaced=False, findNeighbors=False)
    Creates a theta array where every row is a theta vector corresponding to a particle
    uniformSamplingRate gives the number of points for the 0,2pi angle while uniformSamplingRate/2 gives the
    total for all other axes

Parameters
    • numThDim (int) – Number of dimensions of the theta vector
    • uniformSamplingRate (int) – uniformSamplingRate gives the number of points for
      the 0,2pi angle while uniformSamplingRate/2 gives the total for all other axes
    • uniformlySpaced (bool, optional) – Boolean to determine whether or not to try
      and uniformly space the particles on the sphere
    • findNeighbors (bool, optional) – If true, the neighbors (of each theta) will be
      created and returned
Returns

- **th** *(array_like, shape (numParticles,numThDim)) – Output theta array where every row is a different theta vector
- **numParticles** *(int) – Total number of particles. This is also the number of rows of th
- **neighborsList** *(list of list) – List of every particle’s neighbors. It’s a list where every index corresponds to a different particle. The ith index of neighborsList is a list with all the neighbor indices of particle/theta i
- **neighborsLedger** *(list of list) – List of all unique neighbor pairs Every index of this list corresponds to a unique neighbor pair given by [p_iID, p_jID] for every neighboring i,j

**utilities.generateThetaArray_Box** *(subspaceDim, findNeighbors=False)*

Creates a theta array where every row is a theta vector corresponding to a particle and create angles that correspond to bounding box

Parameters

- **subspaceDim** *(int) – Number of dimensions of subspace you need to sample. This is equal to the number of dimensions of theta plus one
- **findNeighbors** *(bool, optional) – If true, the neighbors (of each theta) will be created and returned

Returns

- **th** *(array_like, shape (numParticles,numThDim)) – Output theta array where every row is a different theta vector
- **numParticles** *(int) – Total number of particles. This is also the number of rows of th
- **neighborsList** *(list of list) – List of every particle’s neighbors. It’s a list where every index corresponds to a different particle. The ith index of neighborsList is a list with all the neighbor indices of particle/theta i
- **neighborsLedger** *(list of list) – List of all unique neighbor pairs Every index of this list corresponds to a unique neighbor pair given by [p_iID, p_jID] for every neighboring i,j

**utilities.generateThetaArray_Octagonal** *(subspaceDim, findNeighbors=False)*

Creates a theta array where every row is a theta vector corresponding to a particle and create angles that correspond to octagonal region (box sampling plus directions that bisect each pair)

Parameters

- **subspaceDim** *(int) – Number of dimensions of subspace you need to sample. This is equal to the number of dimensions of theta plus one
- **findNeighbors** *(bool, optional) – If true, the neighbors (of each theta) will be created and returned

Returns

- **th** *(array_like, shape (numParticles,numThDim)) – Output theta array where every row is a different theta vector
- **numParticles** *(int) – Total number of particles. This is also the number of rows of th
- **neighborsList** *(list of list) – List of every particle’s neighbors. It’s a list where every index corresponds to a different particle. The ith index of neighborsList is a list with all the neighbor indices of particle/theta i
- **neighborsLedger** *(list of list) – List of all unique neighbor pairs Every index of this list corresponds to a unique neighbor pair given by [p_iID, p_jID] for every neighboring i,j
utilities.theta2ds (theta)
    Convert theta vector (in radians) to a search direction

utilities.ds2theta (x, twoDBool=False)
    Convert search direction vector to theta vector (in radians) if 2DBool set, then we treat ds vector as if it’s 2D
    because nD has multiple theta vectors for a single ds vector

utilities.dds_dth (th)
    Compute jacobian matrix for d_ds/d_th
    Make sure theta is in radians

utilities.findCubicMin (f0, f1, fp0, fp1)
    Solves for cubic fit minimum given s=0 corresponds to f0, fp0 and s=1 corresponds to f1, fp1
    Cubic fit, f = a + b*s + 1/2*c*s**2 + 1/6*d*s**3

utilities.findCubicFit (f0, f1, fp0, fp1)
    Solves for cubic fit coefficients given s=0 corresponds to f0, fp0 and s=1 corresponds to f1, fp1
    Cubic fit, f = a + b*s + 1/2*c*s**2 + 1/6*d*s**3

utilities.abs_smooth (x, epsi=0.0001)
    Smooth approximation to absolute value function of array

utilities.sign_smooth (x, epsi=0.0001)
    Smooth approximation to sign function of array

utilities.kron_axis (A, B)
    kron(A,B) with A,B square

utilities.dot_kron2 (A, flatB)
    Returns A.dot(B).ravel() or A.dot(B).flatten() where A is matrix and B is inverse vec of flatB
    Uses ROW MAJOR vec and inverse vec operator

utilities.my_minkowski (u, v, p)
    My own version of the scipy version to compute the Minkowski distance of order p between u and v without
    checks

utilities.my_minkowski2 (u, v, p)
    My own version of the scipy version to compute the Minkowski distance of order p between u and v without
    checks

utilities.skewMat (v)
    Takes in a sympy vector as a Matrix object, list of 3 numbers, or numpy array and returns a sympy object that
    represents the skew-symmetric matrix

    Parameters  x (sympy Matrix object, list, numpy array) – Input vector
    Returns  xTilde – Symbolic expression of skew-symmetric matrix of input vector x
    Return type  sympy expression
9.1 Viscous Damper Linear System

In this example, we will look at the case of an object subject to linear damping on its velocity.

9.1.1 Problem Setup

\[
\dot{x} = -\mu \dot{x} + u, \quad \dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & -\mu \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u
\]  

(9.1)

With control input constraints given by

\[|u| \leq 1\]  

(9.2)

and initial condition constraint given by

\[g(x_0, t_0) = \|Mx\|_s - 1 \leq 0\]

(9.3)

9.1.2 Imports

```python
import numpy as np
from dynSystems import LTIDynamics
from constraints import Constraint
from reach import Reachability
from plotting import plotReachGrowth
```

9.1.3 Create Dynamic System Object

```python
mu = 1.0
A = np.array([[0, 1], [0.0, -mu]])
B = np.array([[0.0], [1]])
um = 1.0
LTI = LTIDynamics(A, B, uLimit=um)
```
9.1.4 Create Initial Condition Constraint

\[ E = \text{np.diag}([0.5, 0.1]) \]
\[ IC = \text{Constraint.fromShapeMatrix}(E, p=8) \]

9.1.5 Create Reachability Objects and Initialize Particles

\[ \text{R}_{\text{FRS}} = \text{Reachability}(\text{LTI}, IC) \]
\[ \text{R}_{\text{FRS}}.\text{initializeReach}(\text{subspaceDim}=2, \text{uniformSamplingRate}=60) \]
\[ \text{R}_{\text{BRS}} = \text{Reachability}(\text{LTI}, IC) \]
\[ \text{R}_{\text{BRS}}.\text{initializeReach}(\text{subspaceDim}=2, \text{uniformSamplingRate}=60) \]

9.1.6 Compute Forward Reachable Set over Time Horizon

\[ \text{timeSteps} = 50 \]
\[ t0 = 0.0 \]
\[ tf = 2.0 \]
\[ \text{Tvec} = \text{np.linspace}(t0, tf, \text{timeSteps}) \]
\[ \text{R}_{\text{FRS}}.\text{computeReach}(\text{Tvec}) \]

9.1.7 Compute Trajectories

\[ \text{R}_{\text{FRS}}.\text{computeCurrentStateTrajectories}(\text{numTimeSteps} = \text{timeSteps}) \]
\[ \text{R}_{\text{FRS}}.\text{computeFinalStateHistories}() \]

9.1.8 Convert to Forward Reachable Tube

\[ \text{R}_{\text{FRT}} = \text{R}_{\text{FRS}}.\text{convertReachSetToTube}(\text{returnNew} = \text{True}) \]

9.1.9 Compute Backward Reachable Set over Time Horizon

\[ \text{timeSteps} = 50 \]
\[ t0 = 0.0 \]
\[ tf = -2.0 \ # \text{equivalent to} \ t0 = 2.0, tf = 0.0 \]
\[ \text{Tvec} = \text{np.linspace}(t0, tf, \text{timeSteps}) \]
\[ \text{R}_{\text{BRS}}.\text{computeReach}(\text{Tvec}) \]

9.1.10 Compute Trajectories

\[ \text{R}_{\text{BRS}}.\text{computeCurrentStateTrajectories}(\text{numTimeSteps} = \text{timeSteps}) \]
\[ \text{R}_{\text{BRS}}.\text{computeFinalStateHistories}() \]
9.1.11 Convert to Backward Reachable Tube

\[ R_{BRT} = R_{BRS}.convertReachSetToTube(returnNew=True) \]

9.1.12 Plot results

\[ fh1 = plotReachGrowth(R_{FRS}.particles) \]
\[ fh2 = plotReachGrowth(R_{FRT}.particles) \]
\[ fh3 = plotReachGrowth(R_{BRS}.particles) \]
\[ fh4 = plotReachGrowth(R_{BRT}.particles) \]

9.2 Zermelo’s Problem - Union of Initial Condition Sets

In this example, we will look an instance of the union of multiple independent initial condition sets applied to the Zermelo’s problem.

9.2.1 Problem Setup

\[
\begin{align*}
\dot{x} &= V_m \cos(\theta) + w_x(x, y) \\
\dot{y} &= V_m \sin(\theta) + w_y(x, y) \\
V_m &= 1 \\
w_x &= 1 \\
w_y &= x^2
\end{align*}
\]

(9.4)

With control input constraints given by

\[ \theta \in [0, 2\pi) \]

(9.5)

and initial condition constraints are given by

\[
\begin{align*}
g_1(x_0) &= 9(x_0 + 1)^2 + 100y_0^2 \leq 1 \\
g_2(x_0) &= 100(x_0 + 1)^2 + 9y_0^2 \leq 1
\end{align*}
\]

(9.6)

9.2.2 Create Nonlinear Dynamics Model Using Python Symbolic Toolbox

Imports

```python
from sympy import symbols, Matrix, simplify, eye, sin, cos, zeros, flatten, atan2
from sympy.utilities.autowrap import autowrap
from utilities import cpickleSave
from utilities import sympyNorm, sympyAbsApprox, sympySignApprox
import numpy as np
```

Define Variables, State, and Costate
(x1, x2) = symbols('x1, x2', real = True)  # State variables
(p1, p2) = symbols('p1, p2', real = True)  # Costates variables
(t, ts) = symbols('t, ts', real = True)  # Time variables
N = 2  # size of state and costate
m = 1  # size of control input

# Define state and costate vector
x = Matrix([x1, x2])
# Define costate vector
p = Matrix([p1, p2])
# Define state transition matrix
Phisize = 4*(N**2)
Phi_vars = symbols('Phi:'+str(4*(N**2)))
Phi_mat = Matrix(2*N,2*N,Phi_vars)

Define Optimal control

# Define optimal control policy (if it's analytically possible)
u = atan2(ts*p2,ts*p1)  # analytic uStar

Define State Dynamics

# Define state dynamics
wx = 1.0  # x-direction wind function as a function of x,y coordinates
wy = x1**2.0  # y-direction wind function as a function of x,y coordinates
V = 1.0
xd = Matrix([V*cos(u) + wx, V*sin(u) + wy])
f = xd

From this point, one can use the Symbolic Toolbox derivation template provided in the SCoRe toolbox to generate callable Python functions that represent the state, costate, and trajectory flow state transition matrix dynamics.

9.2.3 Perform Reachability Analysis

Imports

import numpy as np
from dynSystems import SympyDynamicsSmooth
from constraints import Constraint
from reach import Reachability
from plotting import plotReachTrajectories, plotReachVolume, plotReachHistories1D, ...
...plotReachGrowth, plotUnion, plotIntersection

Create Dynamic System Object

Sym = SympyDynamicsSmooth(n=2,fname='zermelo.modname')
Create Initial Condition Constraints

IC1 = Constraint.fromLimits(np.array([1.0/3.0,0.1]),p=2,xc=np.array([-1.0,0.0]))
IC2 = Constraint.fromLimits(np.array([0.1,1.0/3.0]),p=2,xc=np.array([-1.0,0.0]))

Create Reachability Objects and Initialize Particles

R = Reachability(Sym,IC1,jsonLoadFileName='zermelo_p1.json')
R2 = Reachability(Sym,IC2,jsonLoadFileName='zermelo_p1.json')
R.initializeReach(subspaceDim=2, uniformSamplingRate=40)
R2.initializeReach(subspaceDim=2, uniformSamplingRate=40)

Compute Forward Reachable Sets over Time Horizon

timeSteps = 50
t0 = 0.0
tf = 1.0
Tvec = np.linspace(t0,tf,timeSteps)
R.computeReach(Tvec,contMethodOption=2,newtonsCorrectionBool=True)
R.computeCurrentStateTrajectories(numTimeSteps=50)
R.computeFinalStateHistories()
R_T = R.convertReachSetToTube()

R2.computeReach(Tvec,contMethodOption=2,newtonsCorrectionBool=True)
R2.computeCurrentStateTrajectories(numTimeSteps=50)
R2.computeFinalStateHistories()
R2_T = R2.convertReachSetToTube()

Plot results

fh1 = plotReachTrajectories(R.particles)
fh2 = plotReachVolume(R.particles)
fh3 = plotReachGrowth(R.particles,numSnaps=9)
fh4 = plotReachTrajectories(R_T.particles)
fh5 = plotReachVolume(R_T.particles)
fh6 = plotReachGrowth(R_T.particles,numSnaps=9)

fh7 = plotReachTrajectories(R2.particles)
fh8 = plotReachVolume(R2.particles)
fh9 = plotReachGrowth(R2.particles,numSnaps=9)
fh10 = plotReachTrajectories(R2_T.particles)
fh11 = plotReachVolume(R2_T.particles)
fh12 = plotReachGrowth(R2_T.particles,numSnaps=9)

fh13, unionParticles = plotUnion(R,R2)
fh14, intersectParticles = plotIntersection(R,R2)

fh15 = plotReachTrajectories(unionParticles)
fh16 = plotReachVolume(unionParticles)
fh17 = plotReachGrowth(unionParticles,numSnaps=9)
9.3 Duffing Oscillator - Mesh Refinement

In this example, we will look at an instance of performing a refinement of the distribution of particles that constitute the boundary of the reachable set.

9.3.1 Problem Setup

\[ \begin{align*}
    m_1 \ddot{x}_1 &= -k_{1,1}x_1 - k_{1,3}x_3^3 - f_1\dot{x}_1 \\
    &+ k_{2,1}(x_2 - x_1) + k_{2,3}(x_2 - x_1)^3 + f_2(\dot{x}_2 - \dot{x}_1) \\
    m_2 \ddot{x}_2 &= -k_{2,1}(x_2 - x_1) - k_{2,3}(x_2 - x_1)^3 - f_2(\dot{x}_2 - \dot{x}_1) \\
    &+ k_{3,1}(x_3 - x_2) + k_{3,3}(x_3 - x_2)^3 + f_3(\dot{x}_3 - \dot{x}_2) \\
    m_3 \ddot{x}_3 &= -k_{3,1}(x_3 - x_2) - k_{3,3}(x_3 - x_2)^3 \\
    &- f_3(\dot{x}_3 - \dot{x}_2) + u \\
    m_1 &= m_2 = m_3 = 1 \\
    k_{1,1} &= k_{2,1} = k_{3,1} = 1 \\
    k_{1,3} &= k_{2,3} = k_{3,3} = 1/9 \\
    f_1 &= f_2 = f_3 = 1 \\
    u_m &= 1, T_f = \pi
\end{align*} \]

(9.7) (9.8) (9.9)

With control input constraints given by

\[ |u| \leq 1 \]

(9.10)

and initial condition constraints are given by

\[ g(x_0) = x_0^T x_0 - 1 = 0 \]

(9.11) (9.12)

9.3.2 Create Nonlinear Dynamics Model Using Python Symbolic Toolbox

Imports

```python
from sympy import symbols, Matrix, simplify, eye, zeros, flatten
from sympy.utilities.autowrap import autowrap
from utilities import cpickleSave
from utilities import symyNorm, symyAbsApprox, symySignApprox
```

Define Variables, State, and Costate

```python
# Define all variables required for dynamics and include assumptions
(x1, x2, x3, x1d, x2d, x3d) = symbols('x1, x2, x3, x1d, x2d, x3d', real = True)
(p1, p2, p3, pld, p2d, p3d) = symbols('p1, p2, p3, pld, p2d, p3d', real = True)
(t, ts) = symbols('t, ts', real = True) # Time variables
N = 6 # size of state and costate
m = 1 # size of control input

# Define state and costate vector (Matrix with single list input returns a column vector)
x = Matrix([[x3, x3d, x1, x2, x1d, x2d]])
```

(continues on next page)
# Define costate vector
\[
p = \text{Matrix}([p3, p3d, p1, p2, p1d, p2d])
\]
# Define state transition matrix
\[
\text{Phisize} = 4 \times (N^2)
\]
\[
\Phi_\text{vars} = \text{symbols('Phi': str(4 \times (N^2))})
\]
\[
\Phi_\text{mat} = \text{Matrix}(2 \times N, 2 \times N, \Phi_\text{vars})
\]

**Define Optimal control**

\[
\# \text{Define optimal control policy (if it's analytically possible)}
\]
\[
\text{uLimit} = [1.0] \quad \# \max \text{ possible input in each control dimension}
\]
\[
\text{Minv} = \text{eye}(m) \quad \# \text{initialize} \quad \text{inv}(M) \quad \text{matrix}
\]
\[
\text{for } i \text{ in range}(m):
\quad \text{Minv}[i,i] = \text{uLimit}[i] \quad \# \text{place uLimit along diagonal}
\]
\[
\text{B} = \text{Matrix}([[0], [1], [0], [0], [0], [0]])
\]
\[
\text{y} = \text{Minv} \times \text{B} \times \text{p}
\]
\[
\text{pNorm} = 2.0 \quad \# \text{for second option, pNorm is always equal to 2}
\]
\[
\text{s} = 0.0
\]
\[
\text{s2} = \text{sympyNorm}(y)^{2.0}
\]
\[
\text{for } i \text{ in range}(m):
\quad \text{s} += \text{sympyAbsApprox}(y[i], \text{scalar=True})^{(\text{pNorm} / (\text{pNorm}-1.0))}
\]
\[
\text{uTilde} = \text{zeros}(m, 1)
\]
\[
\text{for } i \text{ in range}(m):
\quad \text{uTilde}[i] = \text{sympySignApprox}(y[i], \text{scalar=True}) \times \text{sympyAbsApprox}(y[i], \text{scalar=True})^{(1.0 / (\text{pNorm}-1.0))} / (\text{s}^{(1.0 / \text{pNorm})})
\]
\[
\text{u} = \text{ts} \times \text{simplify}(\text{Minv} \times \text{uTilde}) \quad \# \text{smooth approximation of uStar/optimal control (no need for switch)}
\]
\[
\text{u2} = \text{ts} \times \text{simplify}(\text{Minv} \times y / \text{sympyNorm}(y)) \quad \# \text{analytic uStar for p=2 only (need for switch when s2==0)}
\]

**Define State Dynamics**

\[
m1 = m2 = m3 = 1.0
\]
\[
k11 = k21 = k31 = 1.0
\]
\[
k13 = k23 = k33 = 1.0 / 9.0
\]
\[
f1 = f2 = f3 = 1.0
\]
\[
\# \text{Define state dynamics}
\]
\[
x21 = x2 - x1
\]
\[
x32 = x3 - x2
\]
\[
x21d = x2d - x1d
\]
\[
x32d = x3d - x2d
\]
\[
x1dd = 1 / m1 \times (-k11 \times x1 - k13 \times x1 \times 3.0 - f1 \times x1d + k21 \times x21 + k23 \times x21 \times 3.0 + f2 \times x21d)
\]
\[
x2dd = 1 / m2 \times (-k21 \times x21 - k23 \times x21 \times 3.0 - f2 \times x21d + k31 \times x32 + k33 \times x32 \times 3.0 + f3 \times x32d)
\]
\[
x3dd = 1 / m3 \times (-k31 \times x32 - k33 \times x32 \times 3.0 - f3 \times x32d)
\]
\[
xd = \text{Matrix}([[x3d, x3dd, x1d, x2d, x1dd, x2dd]])
\]
\[
f = xd + B \times u
\]
\[
f2 = xd + B \times u2
\]

From this point, one can use the Symbolic Toolbox derivation template provided in the SCoRe toolbox to generate callable Python functions that represent the state, costate, and trajectory flow state transition matrix dynamics.

9.3. **Duffing Oscillator - Mesh Refinement**
9.3.3 Perform Reachability Analysis

Imports

```python
import numpy as np
from dynSystems import SympyDynamicsSmooth, SympyDynamicsSwitch
from constraints import Constraint
from reach import Reachability
from plotting import plotReachTrajectories, plotReachVolume, plotReachHistories1D,
    plotReachGrowth, plotRedistributionCDFs
from copy import deepcopy
```

Create Dynamic System Object

```python
Sym1 = SympyDynamicsSmooth(n=6,fname='duffing_x3_option1.modname')
Sym2 = SympyDynamicsSwitch(n=6,fname='duffing_x3_option2.modname')
```

Create Initial Condition Constraints

```python
IC = Constraint(M=np.eye(6),p=2)
```

Create Reachability Objects and Initialize Particles

```python
R2 = Reachability(Sym2,IC)
R2.initializeReach(subspaceDim=2,uniformSamplingRate=30)
```

Compute Forward Reachable Sets over Time Horizon

```python
timeSteps = 50
t0 = 0.0
tf = np.pi
Tvec = np.linspace(t0,tf,timeSteps)
R2.computeReach(Tvec,contMethodOption=2)
R2.computeCurrentStateTrajectories(numTimeSteps=timeSteps)
R2.computeFinalStateHistories()
```

Perform Cubic Minimization Mesh Refinement

```python
R2_cubic = deepcopy(R2)
R2.meshRefinement(method='bisect',cost='J',q1=25,q3=65,maxRefinements=3)
R2_cubic.meshRefinement(method='cubic',cost='J',q1=25,q3=65,maxRefinements=3)
```

Plot results
9.4 Cislunar Problem - Reachability with Minimum Control Effort Cost

In this example, we will look an instance of the performing Reachability analyses on systems where one of the variables corresponds to an optimal control objective function.

9.4.1 Problem Setup

\[
\begin{align*}
X &= [x \ y \ \overset{o}{x} \ \overset{o}{y} \ J_1]^T \\
\overset{o}{x} &= 2\overset{o}{y} + x - (1 - \mu)\frac{x - x_1}{\rho_1} - \mu\frac{x - x_2}{\rho_2} + \bar{u}_x \\
\overset{o}{y} &= -2\overset{o}{x} + \left(1 - \frac{1 - \mu}{\rho_1} - \frac{\mu}{\rho_2}\right)y + \bar{u}_y \\
J_1 &= -\int_0^T \|\bar{u}\|_1 \, d\tau = -\int_0^T \bar{u}_x + \bar{u}_y \, d\tau
\end{align*}
\]

(9.13)

With control input constraints given by

\[
|u_x| \leq 5.714e - 5 \, m/s2 \\
|u_y| \leq 5.714e - 5 \, m/s2
\]

(9.14)

and initial condition constraints are given by

\[
g(X_0) = \left(\frac{x - x_{11}}{\epsilon_1}\right)^2 + \left(\frac{y}{\epsilon_1}\right)^2 + \left(\frac{\overset{o}{x}}{\epsilon_2}\right)^2 + \left(\frac{\overset{o}{y}}{\epsilon_2}\right)^2 + \left(\frac{J_1}{\epsilon_2}\right)^2 = 0
\]

\[
\epsilon_1 = 1e - 3 \\
\epsilon_2 = 1e - 4
\]

(9.15)

9.4.2 Create Nonlinear Dynamics Model Using Python Symbolic Toolbox

Imports

```python
from sympy import symbols, Matrix, simplify, eye, sin, cos, zeros, flatten, pi, sqrt
from sympy.utilities.autowrap import autowrap
from utilities import cpickleSave
from utilities import sympyNorm, sympyAbsApprox, sympySignApprox, sympySatApprox
```
Define Variables, State, and Costate

```python
# Define all variables required for dynamics and include assumptions
(J1, x1, x2, x1d, x2d) = symbols('J1, x1, x2, x1d, x2d', real=True) # State variables
(pJ1, px1, px2, px1d, px2d) = symbols('pJ1, px1, px2, px1d, px2d', real=True) # Costates variables
(t, ts) = symbols('t, ts', real=True) # Time variables
N = 5 # Size of state and costate
m = 2 # Size of control input

# Define state and costate vector (Matrix with single list input returns a column vector)
x = Matrix([J1, x1, x2, x1d, x2d])
# Define costate vector
p = Matrix([pJ1, px1, px2, px1d, px2d])
# Define state transition matrix
Phisize = 4*(N**2)
Phi_vars = symbols('Phi:'+str(4*(N**2)))
Phi_mat = Matrix(2*N,2*N,Phi_vars)

# Define problem parameters
Tmax = (0.8e-3)/14 # m/s^2, max thrust [Lunar Icecube from Bosanac paper, 14kg and 0.8mN thrust]
r12 = 384402e3 # m, distance between m1 and m2
G = 6.67430e-11 # m^3/kg/s^2, Newton's gravity constant
m1 = 5.97237e24 # kg, mass of the bigger body
m2 = 7.342e22 # kg, mass of smaller body
controlScale = (r12**2)/(G*(m1+m2)) # conversion from SI to dimensionless units
Tmax = Tmax*controlScale # dimensionless max thrust
mu = 1/(81.3 + 1) # normalized mass ratio for CR3BP - Moon Earth
x1_ = -mu
x2_ = 1-mu
rho1 = sqrt((x1-x1_)**2 + x2**2)
rho2 = sqrt((x1-x2_)**2 + x2**2)
# Lagrange points x-coordinates
L1 = 0.836915
L2 = 1.15568
L3 = -1.00506

Define Optimal control

```
Define State Dynamics

\[
\begin{align*}
x_{1dd} &= 2x_{2d} + x_1 - (1 - \mu)/(\rho_1^{**3}) - \mu/(\rho_2^{**3}) \times x_1 \\
x_{2dd} &= -2x_{1d} + (1 - (1 - \mu)/(\rho_1^{**3}) - \mu/(\rho_2^{**3})) \times x_2 \\
J_{d_L1} &= \text{Matrix}([-u_{L1}\_sum]) \\
x_{d_L1} &= J_{d_L1}\_col\_join(\text{Matrix}([x_{1d}, x_{2d}, x_{1dd}, x_{2dd}])) \\
f &= x_{d_L1} + B\times u_2
\end{align*}
\]

From this point, one can use the Symbolic Toolbox derivation template provided in the SCoRe toolbox to generate callable Python functions that represent the state, costate, and trajectory flow state transition matrix dynamics.

9.4.3 Perform Reachability Analysis

Imports

```python
import numpy as np
from dynSystems import SympyDynamicsSmooth, SympyDynamicsSwitch
from constraints import Constraint
from reach import Reachability
from plotting import plotReachTrajectories, plotReachVolume, plotReachHistories1D,
                  plotReachGrowth
```

Create Dynamic System Object

```python
Sym2 = SympyDynamicsSmooth(n=5, fname='cislunar2_option2.modname')
```

Create Initial Condition Constraints

```python
L1 = 0.836915
xc = np.zeros(5)
xc[1] = L1
IC = Constraint.fromLimits(np.array([0.0001, 0.001, 0.001, 0.0001, 0.0001]), xc=xc, p=2)
```

Create Reachability Objects and Initialize Particles

```python
R = Reachability(Sym2, IC)
R.initializeReach(subspaceDim=3, uniformSamplingRate=30)
```

Compute Forward Reachable Sets over Time Horizon

```python
timeSteps = 50
t0 = 0.0
numDays = 5 # number of days to propagate
tf = (2.0 * np.pi / 27.322) * numDays
Tvec = np.linspace(t0, tf, timeSteps)
R.computeReach(Tvec, contMethodOption=1, printProgress=True, newtonsCorrectionBool=True)
R.computeCurrentStateTrajectories(numTimeSteps=30)
R.computeFinalStateHistories()
```
Plot results

\[ w = 2.6653296644361014 \times 10^{-6} \] # angular velocity of EM system in rad/s

\[ r_{12} = 384402000.0 \] # distance from Earth to moon in meters

\[ r_{12\_km} = \frac{r_{12}}{1000} \] # distance from Earth to moon in kilometers

\[ \text{nonDim2SI2} = w \cdot r_{12} \] # converts \( J_1 \) to \( \Delta v \) in m/s

# Plot results!

\[ \text{fh1} = \text{plotReachTrajectories}(R\text{.particles, projDim=[2, 3, 1], axisEqualBool=F\text{alse}, sc=[r_{12\_km}, r_{12\_km}, \text{nonDim2SI2}], axesLabels=['x [km]', 'y [km]', '$J_1$ [m/s]']) } 

\[ \text{fh2} = \text{plotReachTrajectories}(R\text{.particles, projDim=[3, 1], axisEqualBool=F\text{alse}, sc=[r_{12\_km}, \text{nonDim2SI2}], axesLabels=['x [km]', '$y$ [km]', '$J_1$ [m/s]']) } 

\[ \text{fh3} = \text{plotReachTrajectories}(R\text{.particles, projDim=[2, 1], axisEqualBool=F\text{alse}, sc=[r_{12\_km}, \text{nonDim2SI2}], axesLabels=['x [km]', '$y$ [km]', '$J_1$ [m/s]']) } 

\[ \text{fh4} = \text{plotReachVolume}(R\text{.particles, projDim=[2, 3, 1], axisEqualBool=F\text{alse}, sc=[r_{12\_km}, r_{12\_km}, \text{nonDim2SI2}], axesLabels=['x [km]', 'y [km]', '$J_1$ [m/s]']) } 

\[ \text{fh5} = \text{plotReachVolume}(R\text{.particles, projDim=[2, 3], axisEqualBool=F\text{alse}, sc=[r_{12\_km}, r_{12\_km}], axesLabels=['x [km]', 'y [km]', '$J_1$ [m/s]']) } 

\[ \text{fh6} = \text{plotReachHistories1D}(R\text{.particles, projDim=1, shadedTrajBool=T\text{rue, particleHistBool=T\text{rue, sc=nonDim2SI2}, axesLabels=['T', '$J_1$ [m/s]']) } 

Save/Export Results

\[ \text{vArr1, pArr1 = R.vertices('xf_T', T\text{rue})} \]

\[ \text{vArr2, pArr2 = R.vertices('x', T\text{rue})} \]

\[ \text{vArr3, pArr3 = R.vertices_over_T(T\text{rue})} \]

\text{from scipy.io import savemat}

\text{savemat('cislunarWayMoreParticles5_over_T_Again.mat', { 'vArr1': vArr1, 'pArr1': pArr1, 'vArr2': vArr2, 'pArr2': pArr2, 'vArr3': vArr3, 'pArr3': pArr3, 'Tvec': R.Tvec, })}
For an more detailed explanation of particular concepts, please refer to:

Using Continuation Methods to Compute Convex Reachable Volume Projections
Brew Thesis Proposal
Decentralized Techniques for Sampling Boundary of Subspace Reachable Set
Connections with Reachability Theory and Multi-objective Optimization
CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search
PYTHON MODULE INDEX

c
constraints, 32
contMethodMod, 52

d
dynSystems, 37

p
particle, 29
plotting, 53

q
qLPV, 50

r
reach, 23

u
utilities, 59
INDEX

Symbols

__init__() (constraints.Constraint method), 33
__init__() (dynSystems.LTIDynamics method), 37
__init__() (dynSystems.LTVDynamics method), 40
__init__() (dynSystems.SympyDynamicsSmooth method), 43
__init__() (dynSystems.SympyDynamicsSwitch method), 45
__init__() (dynSystems.dynSystemTemplate method), 47
__init__() (particle.Particle method), 30
__init__() (qLPV.qLPV method), 30
__init__() (reach.Reachability method), 23

A

A() (qLPV.qLPV method), 51
abs_smooth() (in module utilities), 64
addParticles() (reach.Reachability method), 26
adj() (in module utilities), 60
angBtwnVectors() (in module utilities), 59
angleBtwnUnitVectors() (in module utilities), 59
arcLengthEvent1() (in module contMethodMod), 53
arcLengthEvent1s() (in module contMethodMod), 53
arcLengthEvent2() (in module contMethodMod), 53
arcLengthEvent3() (in module contMethodMod), 53
arcLengthEvent4() (in module contMethodMod), 53
axisEqual() (in module plotting), 53

B

B() (qLPV.qLPV method), 51

computeCurrentStateTrajectories() (reach.Reachability method), 25
computeDistanceCost() (particle.Particle method), 32

computeFinalStateHistories() (reach.Reachability method), 25
computeGraphDistanceCost() (particle.Particle method), 32
computeReach() (reach.Reachability method), 24
computeTangentVector() (in module contMethodMod), 53
computeTrajectoriesOverT() (reach.Reachability method), 25

Constraint (class in constraints), 32
costs (module), 32
contMethod() (in module contMethodMod), 53
contMethod_T() (in module contMethodMod), 52
contMethod_toT() (in module contMethodMod), 53
contMethodMod (module), 52
contMethodODE() (in module contMethodMod), 52
contMethodODE_arcLength() (in module contMethodMod), 53
contMethodODE_arcLength_generalized() (in module contMethodMod), 53
contMethodODE_arcLength_Sard() (in module contMethodMod), 53
convertReachSetToTube() (reach.Reachability method), 28
cpickleLoad() (in module utilities), 62
cpickleSave() (in module utilities), 62
createBallTree() (reach.Reachability method), 26
createMultiDimList() (in module utilities), 62

d

d2g() (constraints.Constraint method), 34
d3g() (constraints.Constraint method), 35
dds_dth() (in module utilities), 64
dF_dds() (particle.Particle method), 32
dF_dth() (particle.Particle method), 31
dfdp() (qLPV.qLPV method), 51
dFDt() (particle.Particle attribute), 29
dfx() (qLPV.qLPV method), 51
dFdz() (particle.Particle attribute), 29
dg() (constraints.Constraint method), 34
dgd() (qLPV.qLPV method), 51
dgdx() (qLPV.qLPV method), 51
Index

84

dillSave() (in module utilities), 62
dillLoad() (in module utilities), 62
dot_kron2() (in module utilities), 64
dpdt() (dynSystems.dyntSystemTemplate method), 48
dpdt() (dynSystems.LTIDynamics method), 38
dpdt() (dynSystems.LTVDynamics method), 41
dpdt() (dynSystems.SympyDynamicsSmooth method), 44
dpdt() (dynSystems.SympyDynamicsSwitch method), 46
dPhidt() (qLPV.qLPV method), 51
ds2thetatheta() (in module utilities), 64
dxdt() (dynSystems.dyntSystemTemplate method), 48
dxdt() (dynSystems.LTIDynamics method), 38
dxdt() (dynSystems.LTVDynamics method), 41
dYdt() (dynSystems.SympyDynamicsSmooth method), 43
dYdt() (dynSystems.SympyDynamicsSwitch method), 46

F

F() (particle.Particle method), 32
findCubicFit() (in module utilities), 64
findCubicMin() (in module utilities), 64

FNorm() (particle.Particle method), 32
fromds() (particle.Particle class method), 30
fromLimits() (constraints.Constraint class method), 33
fromShapeMatrix() (constraints.Constraint class method), 33

G

g() (constraints.Constraint method), 34
generateThetaArray() (in module utilities), 62
generateThetaArray_Box() (in module utilities), 63
generateThetaArray_Octagonal() (in module utilities), 63

H

H() (dynSystems.dyntSystemTemplate method), 48
H() (dynSystems.LTIDynamics method), 39
H() (dynSystems.LTVDynamics method), 41
H() (dynSystems.SympyDynamicsSmooth method), 44
H() (dynSystems.SympyDynamicsSwitch method), 46

I

initializeReach() (reach.Reachability method), 24

K

kron_axis() (in module utilities), 64

L

LTIDynamics (class in dynSystems), 37
LTVDynamics (class in dynSystems), 40

M

matNorm() (in module utilities), 59
maxInnerProduct() (constraints.Constraint method), 35
maxInnerProductJacobianNumerical() (constraints.Constraint method), 37
maxInnerProductSmooth() (constraints.Constraint method), 36
maxInnerProductSmoothJacobian() (constraints.Constraint method), 36
maxInnerProductSmoothJacobian2() (constraints.Constraint method), 36
meshRefinement() (reach.Reachability method), 26

my_minkowski() (in module utilities), 64
my_minkowski2() (in module utilities), 64

N

neighborsLedger (reach.Reachability attribute), 23
T
theta2ds() (in module utilities), 63
trimUpdate() (qLPV.qLPV method), 51

U
update_jacobian() (qLPV.qLPV method), 51
update_qLPV() (qLPV.qLPV method), 51
updateDistanceCost() (reach.Reachability method), 26
updateds() (particle.Particle method), 30
updateFinalStates() (particle.Particle method), 31
updateFinalStates_y() (particle.Particle method), 31
updateGraphDistanceCost() (reach.Reachability method), 26
updateReach() (reach.Reachability method), 26
updateth() (particle.Particle method), 30
updatey() (particle.Particle method), 31
updatez() (particle.Particle method), 30
uStar() (dynSystems.LTIDynamics method), 38
uStar() (dynSystems.LTVDynamics method), 40
uStar() (qLPV.qLPV method), 51
utilities (module), 59
utrm() (qLPV.qLPV method), 51

V
V() (particle.Particle method), 31
vec2radec() (in module utilities), 60
vec2radec_rad() (in module utilities), 60
vertices() (reach.Reachability method), 27
vertices_over_T() (reach.Reachability method), 28
Vx() (particle.Particle method), 31
Vxx() (particle.Particle method), 31

X
xtrm() (qLPV.qLPV method), 51

Y
y_t_T (particle.Particle attribute), 29

Z
z (particle.Particle attribute), 29