USING SAMPLE-BASED CONTINUATION TECHNIQUES TO EFFICIENTLY COMPUTE SUBSPACE REACHABLE SETS AND PARETO SURFACES

A Thesis
Presented to
The Academic Faculty

By

Julian Brew

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Aerospace Engineering

Georgia Institute of Technology

December 2019

USING SAMPLE-BASED CONTINUATION TECHNIQUES TO EFFICIENTLY COMPUTE SUBSPACE REACHABLE SETS AND PARETO SURFACES

Approved by:

Dr. Marcus J. Holzinger, Advisor
Department of Aerospace Engineering Sciences
University of Colorado Boulder

Dr. E. Glenn Lightsey
School of Aerospace Engineering
Georgia Institute of Technology

Stefan Schuet
Research Engineer
NASA Ames Research Center

Dr. Panagiotis Tsiotras
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Jonathan Rogers
School of Aerospace Engineering
Georgia Institute of Technology

Date Approved: October 28, 2019

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.

Sir William Bragg

To my mom and dad, who taught me that I can do anything

ACKNOWLEDGEMENTS

First I would like to thank Disney. Disney movies have taught me many life lessons growing up and these lessons continue to today. When I was tired and ready to give up this idea of a Ph.D., remembering *Hercules* or *The Princess and the Frog* helped me understand that hard work pays off if you persevere. When my code broke over and over, remembering the mantra of *Meet the Robinsons* reminded me to "Keep Moving Forward". Plus there's always my favorite movie of all time, *A Goofy Movie*, that reminds me that fortune favors the bold.

I would also like to thank my trio of advisors: Marcus Holzinger, Glenn Lightsey, and Stefan Schuet. All of you have helped guide me with research, graduate school, careers, and life in general. Looking back to the beginning of my graduate school career, I know a lot of the personal and academic growth is due to you all.

Of course, I would not have made it through all of the long nights in 209 doing homework, writing papers, and doing random proofs about Lagrange interpolation without my fellow members of the Holzinger research group and the SSDL Trolls. I appreciate the many useful discussions about research problems, but also the much-needed distractions from work like the Counts, Nerf wars, and long discussions about why the *The Last Jedi* is one of the best Star Wars movies. You helped make graduate school a memorable, somewhat tolerable experience for me.

Finally, this research was supported by the NASA Space Technology Research Fellowship (NSTRF) Grant No. NNX16AM39H. This support made this entire journey possible and I'm forever grateful. I would also like to thank Matthias Althoffand Niklas Kochdumper for their assitance with the CORA toolbox.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

For a given continuous-time dynamical system with control input constraints and prescribed state boundary conditions, one can compute the reachable set for a specified time horizon. Forward reachable sets contain all states that can be reached using a feasible control policy at the specified time horizon. Alternatively, backwards reachable sets contain all initial states that can reach the prescribed state boundary condition using a feasible control policy at the specified time horizon. The computation of reachable sets has been applied to problems such as vehicle collision avoidance, operational safety planning, system capability demonstration, economic modeling, and weather forecasting.

While numerous methods have been developed to compute reachable sets, these methods generally make sacrifices in accuracy and computational demand due to considerations such as system dynamics, geometric representation of the sets, and dimensional scalability. The most dimensionally scalable methods use geometric objects such as polytopes, zonotopes, ellipsoids, and support functions with linear assumptions to compute approximations of the reachable set. On the other hand, Hamilton-Jacobi-Bellman reachability formulations are the most general in terms of system dynamics and geometric representation, but suffer from the curse of dimensionality and most are computationally intractable for high-dimensional systems.

The first contribution of this thesis presents a novel method for computing forward/backwards reachable sets/tubes for continuous time nonlinear dynamic systems by efficiently solving for samples of the reachable set boundary using optimal control and continuation methods. This techinique is a compromise between the accuracy and system generality of the Hamilton-Jacobi-Bellman methods with the speed and dimensional scalability of the geometric set-based methods. Furthermore, dimensionally-driven costs can be significantly reduced by only computing projections of the reachable set onto user-specified state di-

mensions.

As the reachable set boundaries are described using independent point solutions, the evolution of these reachable sets over time results in sparse and dense collections of point solutions. The second contribution of this thesis presents analytic results necessary to prove distributed computation convergence, as well as necessary conditions for curvature- or uniform coverage-based sampling methods. For curvature-based sampling, support functions are used to identify regions of the reachable set boundary with high curvature. For distance-based sampling, the proposed approach uses distributed control methods by treating each point solution as a decentralized agent and solving the surface coverage problem. Furthermore, by judiciously spawning additional point solutions, it's possible to increase the sampling uniformity of the reachable set boundary.

The third contribution of this thesis draws connections between reachability theory and multi-objective optimization. Both reachability and multi-objective optimization problems search for surfaces in objective space satisfying the necessary conditions of optimality. Furthermore, reachability surfaces and pareto-optimal surfaces are both sets of extremal solutions that optimize some objective function. This suggests a connection between the two fields with the potential of cross-fertilization of computational techniques and theory.

Numerical demonstrations of the discussed contributions are given. Also, a brief comparison of the methods presented in this thesis with alternate reachability analysis software toolboxes is provided.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Reachable sets are volumes in state-space that can be reached given an integration constraint such as a time horizon and a control constraint. The computation of reachable sets has been applied to problems such as vehicle collision avoidance, operational safety planning, and capability demonstration [1, 2, 3].

The canonical developments of reachability theory are derived from optimal control theory [4, 5, 6]. In these formulations, computing the reachable set for a system involves computing solutions the Hamilton-Jacobi-Bellman partial differential equation (HJB PDE). The zero level sets of the value function over time represent the boundary of the minimum time reachable set [7, 8]. This solution approach has many parallels with computational fluid dynamics (CFD) problems in which space and time are discretized into a grid and the governing Navier-Stokes partial differential equations are solved numerically [7, 9]. This type of solution method is Eulerian because it requires gridding the state space, resulting in time and memory requirements that scale exponentially with the state dimension. Because of this, the computation of reachable sets using Eulerian methods are generally intractable for state space dimensions greater than $n = 4$ due to the curse of dimensionality. An extension of this technique involves computing the evolution of the projection of the overall level set on a subspace of interest [10].

To reduce the computation burden incurred by these HJB methods, alternate techniques have been developed to compute over/under-approximations for the reachable set [11, 12, 13, 14, 15]. These methods generally use pre-defined geometric objects such as polytopes,

zonotopes, ellipsoids, and support functions relying on linear dynamics or conservative linear approximations of nonlinear dynamics [15, 16, 17, 18]. While these methods have demonstrated superior state dimension scalability with reachable sets with hundreds of states, many of them rely on approximating both the system dynamics and reachable set description to user-specified precision [15, 16, 17, 18].

The methods presented in this thesis aim to be a compromise between the accuracy and system generality of the HJB methods with the speed and dimensional scalability of the geometric set-based methods. The presented technique can compute reachable sets for Lipschitz continuous dynamic systems without approximations to the system dynamics as performed in other set-based methods [11, 13, 12, 15, 18]. The set representation is given by samples on the reachable set boundary and first-order surface tangent information. By construction, these samples are exactly (within numerical integration precision) on the convex-hull of the true reachable set as opposed to over/under-approximations given by other methods [11, 13, 15, 17]. While the proposed apporach is approximative due to numerical integration, the accuracy can be user-specified by adjusting the integration error tolerances as well as by performing local root-finding onto the extremal surface described by the necessary conditions of optimality. The methods presented in this thesis are fundamentally different from HJB and set representation methods in that continuation methods are utilized to gather local topology information on the reachable set boundary. Furthermore, the presented method simultaneously computes the extremal trajectories and corresponding optimal control signals that describe the reachable set boundary.

1.2 Thesis Statement

Sampled reachability solutions and Pareto-optimal sets can be tractably computed and distributed along surface manifolds using optimal control and continuation methods.

1.3 Subspace Reachable Sets

In many cases the end user is only interested in a subset of states in a reachability analysis as opposed to the reachable set in the full state space dimension. Because of this, it is useful to only compute the reachable set of the subspace that contains the states of interest. Thus, only the computational cost of the subspace of interest is incurred as opposed to the computational cost of the full description of the reachable set. Holzinger et al. demonstrated application of the transversality conditions on sampled individual trajectories to allow subspace reachable set computation with significantly lower dimensionality-driven costs [19].

In addition, applying the necessary conditions of optimality, dynamics constraints, and initial condition constraints, a point solution of the subspace extremum surface may be used to find nearby solutions. The field of numerical continuation investigates how solutions to parameterized systems of equations change with respect to the above parameters. Numerical continuation methods have applications in the study of dynamical systems particularly in chaotic system analysis, optimal control, parametric bifurcation, and in the search for quasi-periodic invariant tori in the circular restricted three body problem [20, 21, 22]. Using this numerical continuation approach and an initial state that satisfies the constraints, the computation of a point solution on the subspace extremum surface is reduced to an initial value problem solvable through numerical integration. The required computation under this proposed approach exponentially reduces from the problem dimension to the subspace dimension ($\mathcal{O}(k^{n-1}) \rightarrow \mathcal{O}(k^{s-1})$ where $1 \leq s \leq n$). As a result, a large variety of previously intractable reachability problems become computationally feasible.

The technique described in this thesis has a large overlap with the family of support functions often used in reachability analysis [16, 23, 24, 25]. Support functions are a common way of expressing convex sets and are commonly used in convex analysis as they can be used to efficiently describe a large class of sets such as polytopes, zonotopes, and ellipsoids [26, 16]. Furthermore, a support function representation has the advantage that set

operations such as Minkowksi sums, convex hulls, and linear maps can all be implemented efficiently [26]. Instead of set operations, the methodology discussed in this thesis uses optimal control and continuation methods to compute the support functions that represent the reachable set. In this way, nonlinear dynamical systems can be analyzed, optimal trajectories can be computed, and optimal control policies can be generated.

Reachability problems belong to two primary classes- forward and backwards - depending on the time associated with the boundary condition on the system states. For forward reachable sets (FRS), the state boundary condition specifies the states at the initial time that are feasible or reachable. As time increases, the forward reachable set defines all states starting from the boundary condition set that can be achieved at the specified time. For backwards reachable sets (BRS), the state boundary condition specifies the states at the final time that are feasible or reachable. As time decreases, the backwards reachable set defines all initial states that can achieve the boundary condition set at the specified initial time. If the boundary condition set at the final time is deemed unsafe, the backwards reachable set contains unsafe initial states. Thus, backwards reachability analyses are often used for safety assurance, fault detection, and collision avoidance problems [27, 28]. Closely related concepts are reachable tubes, which represent the set of reachable states throughout the entire specified time horizon. Geometrically, the reachable tubes are the union of the reachable sets over the time horizon.

Previous work by Holzinger et al. focused on ellipsoidal and spherical constraints for initial condition and control input, respectively [19, 29]. This contribution introduces a more generalized formulation of these constraints based on affine transformations of unit balls of normed vector spaces of the p-norm or F-norm type [30]. Along with additional generality, these extensions allow for smooth and differentiable approximations to rectangular and box constraints which are commonplace in system verification and reachability analyses.

In addition to using the unit-ball type initial condition set, this contribution introduces

initial condition sets created from the union of multiple initial condition sets. Intuitively, the union of the initial condition sets results in the union of the resulting reachable volumes. Furthermore, as each reachable volume is represented using samples, operations such as intersections, convex hulls, and unions of reachable volumes can be performed relatively easily.

Holzinger and Brew have derived numerical continuation methods based on parametrizing the optimal solution curves as a function of reachability time horizon [19, 31]. The major difficulty in this continuation method is when the Jacobian of the optimality curve with respect to optimal solution approaches singularity or is singular. This signifies that there may not be a unique, one-to-one mapping from the reachability time horizon to the optimal reachability solution. One common approach to handle a number of cases where the Jacobian is singular is to use pseudo-arclength continuation [20]. In this case, the optimal solution curve is parametrized by arclength. This removes the singularity when the rank-defiency of the Jacobian is, at most, one. This contribution describes how pseudo-arclength continuation may be used to compute the optimal solution curves for reachability problems.

Contribution 1 : The formulation of the optimal control policy and continuation method approach for forward and backwards subspace reachability computations using sampling methods.

1.4 Parallelized Distributed Control of Extremal Solutions

When describing a continous surface using discrete samples, there are generally two criteria used to generate samples. One criteria is based on local curvature of the surface. This case is used frequently in computer aided geometric design and graphics fields when representing surfaces because it leads to more efficient reconstruction of the continuous surface from limited points using interpolation [32, 33]. Also, it is desirable in many cases to effectively describe significant features in the surface described by the curvature [33, 34]. Other crite-

ria for generating samples of a surface are based on the uniformity of distances between all the samples. This case is used frequently for rendering and faithfully representing overall surface geometry using a limited number of points [35, 36].

When using distance-based criteria between different samples and their neighboring samples, it is useful to describe this collection of interconnected reachability set samples using a graph. This takes advantage of the fact that samples only need local information from neighboring samples to evaluate a uniformity metric. For computational purposes, minimal communication links are desirable as this negatively impacts the parallelizability and increases computational overhead [37]. Fortunately, distributed control using only local information has been well studied [38, 39, 40]. Developed theory behind multi-agent robotics problems based on Gabriel graphs and central Voronoi tessellations can be applied to this problem to transform each reachability problem into a parallelized distributed control problem where each "agent" is a point solution of the reachability subspace problem [40]. There is currently no common ground between distributed control and reachability set computation.

In addition to the use of distributed control to adjust the position of the extremal point solutions, point solutions may be added or removed from the graph. By prescribing lower and upper bounds on distance metrics between point solutions, detecting sparse or dense regions on the extremal surface can be achieved by identifying point solution pairs that do not satisfy these bounds. As the reachable set evolves, it is then possible to spawn new point solutions and potentially remove redundant point solutions. Either of these changes may occur independently and will each result in a updated graph. Preliminary work used interquartile range (IQR) outlier detection from univariate statistics to select the upper threshold [29].

In addition to the type of sampling criteria used for the description of the reachable set boundary, there is a tradeoff between number of samples and surface resolution. Because

the final reachable set shape and size is not known *a priori*, it's sometimes difficult to specify a fixed number of particles to describe this reachable set boundary. In cases like this, it is possible for the user to explicitly specify a desired spatial resolution for the reachability set as opposed to an implicit specification such as number of samples or facets. The aforementioned technique of adding samples is very convenient in this situation. However, depending on the problem, this could lead to a large increase in required computational memory to compute these samples.

Each of the discussed methods for reachable set boundary exploration only requires the optimal control problem state and costate from the point solution itself. As a result, each numerical continuation method instance is computationally independent from the information from other point solutions. This parallel aspect of the methodology makes available computational techniques suited for parallel and/or distributed computing.

Contribution 2 : The development of distributed control policies to enforce uniform subspace reachable surface coverage while minimizing the required inter-agent communication.

1.5 Connections with Multi-objective Optimization

Multi-objective optimization problems are those in which there are a number of independent objective functions that need to optimized over a set of design variables [41, 42, 43]. In many cases, one seeks to achieve a balance between the given objectives, such as cost or performance. Multi-objective optimization methods are used in a wide range of applications, including engineering system design [44], financial asset allocation [45], spacecraft maneuver design [46], and weather forecasting [47].

Many optimization problems in engineering and economics are inherently multi-objective because there are often tradeoffs in metrics of interest. The result from this optimization

is the Pareto frontier, set of Pareto-optimal points, where it is impossible to change any design variables to improve one objective without degrading another [41]. Once this Pareto frontier is computed, the user can make evaluate tradeoffs between efficient solutions as opposed to the full objective space.

Algorithms for computing Pareto frontiers can be loosely organized into the following classes: iterative single-objective optimization, metaheuristic algorithms, and homotopy methods. In iterative single-objective optimization, a single aggregate objective function is formed using the individual objectives, usually parametrized by relative weights. Methods within this class of algorithms include weighted L_p-metric method, ϵ-constraint method, method of equality constraints, and the normal-boundary intersection method [48, 49, 50, 44]. Metaheuristic algorithms are generally stochastic methods that attempt to efficiently explore the feasible design space to find near-optimal solutions. Common examples of this class of methods include evolutionary algorithms, simulated annealing methods, and particle swarm methods [51, 52, 53]. A final class of algorithms focus on homotopy or continuation. These methods work by computing paths and surfaces in the feasible design space that map to the Pareto-optimal set of points [54, 55, 56, 57].

Both reachability and multi-objective optimization problems search for extremal sets in objective space. For reachability problems, the objective space is the state space for the dynamic variables. Furthermore, both problems involve the search for solutions that satisfy the first-order necessary conditions of optimality [4, 5, 58]. The curse of dimensionality affects research in both fields, limiting applicability of many methods to low-dimension and/or simplified objectives [59, 60, 61]. Moreover, continuation method solution approaches have been applied to both problems [56, 62, 63, 64, 65].

The interconnections between the theory of multi-objective optimization and control theory have been stuided in the past, primarily in path planning and multi-objective optimal control applications [66, 67]. Mitchell and Sastry solved multi-objective path planning problems

using the weighted sum approach of multi-objective optimization along with multiple solutions of a parametrized HJB PDE [68]. Kumar extended this concept by augmenting the dynamic state with integral constraints and solving a single, higher-dimensional HJB PDE [69]. The aim of this contribution is to further investigate the analytic connections between reachability and multi-objective optimization theory. This will be done by creating a joint formulation of optimal control and multi-objective optimization using the generalized independent parameter (GIP) HJB PDE [70, 71]. Furthermore, it is shown that the first-order necessary conditions of the joint formulation reduce to the first-order necessary conditions of both Pareto-optimality and optimal control.

For the most part, reachability analysis and multi-objective optimization problems are contained in different fields. Moreover, the analytical techniques and numerical algorithms to solve these problems vary between the fields. It is the hope of this contribution to help bridge the gap between the two fields. Unification of these separate fields can enable cross-pollination of both theory and numerical methods.

Contribution 3 : The joint formulation between multi-objective optimization and reachability problems to allow for simultaneous computation of Pareto-optimal sets and reachable sets.

1.6 Summary of Contributions and Relevant Literature

To summarize, the contributions of this work aim to develop efficient computational methods for reachability surfaces. The first contribution involves the use of parameterized optimal control and continuation methods to efficiently sample subspace reachable extremal points. The second contribution builds upon the first by utilizing parallelization and distributed control of the extremal point solutions to efficiently compute and render the reachability surface of interest. The third contribution draws connections between reachability theory with multi-objective optimization by computing pareto-optimal surfaces using the

methodology described by the first two contributions. Table 1.1 summarizes the relevant literature for each of the contributions identified.

Table 1.1: Overview of proposed contributions, existing literature and proposed publications

		Contribution 1: Sampling-based subspace reachability	Contribution 2: Parallelized Multi-agent Exploration	Contribution 3: Connections to Multi-objective Optimization
Representative Existing Literature (by reference index)	Varaiya [4]	*Formulation of convex reach set using optimal control*		
	Holzinger and Scheeres [65]	*Optimal control policy and optimality conditions for reachability with ellipsoidal initial sets*		
	Holzinger and Scheeres [23]	*Derivation of subspace reachability using sampling methods*		
	Ohtsuka and Fujii [25]	*Formulation of optimal control problem solution through continuation*		
	Mitchell and Tomlin [10]	*Deriving modified HJB PDE for overapproximate reach set projections*		
	Mesbahi and Egerstedt [40]		*Multi-agent network control*	
	Batalin et. al. [93]		*Multi-robot coverage using local distance metrics*	
	Zomaya [37]		*Parallel and distributed computing principles*	
	Poduri et. al. [39]		*NET graphs for distributed topology control*	
	Holzinger et. al. [70]			*Introduction of generalized independent parameter optimal reachability*
	Hillermeier [55]			*Homotopy algorithm for solution of multiobjective optimization problems*
	Daskilewicz [57]			*Continuation-based algorithm towards uniform sampling of pareto frontiers*
My Publications	[28] - SFM '17	*Formulation of forward and backward reachable set and reachable tube computations using sampling methods*	*Reasch surface description improvement using neighboring point-wise distance metric*	
	[33] - IAC '18	*Implemented numerical conditioning methods to improve accuracy of continuation method*		
	[63] - (Submitted to JGCD '19)			
	(Conference: Spr '20)		*Development of distributed control policies to enforce uniform subspace reachable surface coverage*	
	[64] - (Submitted to JGCD '19)			
	(Conference: Spr '20)			*Analytical connections between reachability theory and multi-objective optimization*
	(Ready to Submit to OIE '19)			

CHAPTER 2

USING CONTINUATION METHODS TO COMPUTE REACHABLE VOLUME PROJECTIONS

2.1 Sampling Methods for Subspace Reachability

An optimal reachability problem can be defined as a continuum of optimal control problems with initial conditions satisfying an inequality constraint on the initial value function $V(\mathbf{x}, t_0) \leq 0$. The optimal control problem is formally stated as

$$\begin{aligned} \max_{\mathbf{u}\in U} \Bigg[\int_{t_0}^{t_f} \mathcal{L}(\mathbf{x}(\tau), \mathbf{u}(\tau), \tau) d\tau + V(\mathbf{x}_f, t_f) \Bigg] \\ s.t. \qquad \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{g}(\mathbf{x}_0, t_0, \mathbf{x}_f, t_f) = \mathbf{0} \end{aligned} \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state, $\mathbf{u} \in \mathbb{R}^m$ is the control input, $t \in [t_0, t_f]$ is time, $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ is the trajectory performance function, $V : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is the terminal performance function, $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n$ captures the system dynamics, $\mathbf{g} : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^v$ expresses boundary conditions, and $U \subseteq \mathbb{R}^m$ defines the set of admissible controls.

Commonly, minimum time reachability analyses are performed. In such case, the Lagrangian term may be set $\mathcal{L}(\mathbf{x}, \mathbf{u}, t) = 0$ [72]. Often minimum time reachability sets are determined by computing viscosity solutions of the HJB PDE where the zero-level sets of V over time represent the boundary of the minimum time reachability set. This solution approach requires space and time to be discretized into grids and the governing partial differential equations solved numerically. While these methods find accurate approximations to the reachable sets, the curse of dimensionality continues to hamper the computation of

the reachable sets as the required computational load typically scales exponentially with the problems state dimension.

One method of avoiding the computational cost of the full state space reachable sets is to only compute the reachable sets in terms of subspaces of the full state space. This is analogous to computing projections of the full state reachable set onto subspaces interest. The methodology is to solve optimal control problems whose solutions compose the boundary of the reachable set. This approach is explained by Holzinger and the highlights of the approach are summarized here [19].

To avoid unnecessary computational cost by calculating the full state reachable set, the reachability computation is performed on a subspace $\mathbb{R}^s$ of the full state space $\mathbb{R}^n$ ($\mathbb{R}^s \subseteq \mathbb{R}^n$). This can be performed by decomposing the full state into the subspace of interest and residual subspace such that $\mathbf{x} = [\mathbf{x}_s^T \ \mathbf{x}_r^T]^T$. To compute maximal reachable sets, the optimal control is derived to make the reachable set size as large as possible [73]. Therefore, to compute points on the subspace reachability set after an amount of time, the final subspace distance must be maximized in a direction of interest in the given subspace [19]. This approach is analogous to the weighted L_p method in multi-objective optimization [43]. A search direction in $\mathbb{R}^s$ can be described using $\hat{\mathbf{d}}_s = \mathbf{h}(\boldsymbol{\theta})$ where $\boldsymbol{\theta} \in \mathbb{R}^{s-1}$ is a vector of angular coordinates and $\mathbf{h}$ is the mapping from the angular coordinates to a cartesian direction vector. For this work, hyperspherical coordinates are used to describe the unit vector search directions.

It is then possible to construct the performance index for a point solution on the subspace reachable set as

$$\begin{aligned} V(\mathbf{x}_f, t_f, \boldsymbol{\theta}) &= \frac{1}{2}\mathbf{x}_f^T G(\boldsymbol{\theta})\mathbf{x}_f \\ G(\boldsymbol{\theta}) &= \begin{bmatrix} \tilde{\boldsymbol{G}}_{s\times s} & \mathbf{0}_{s\times r} \\ \mathbf{0}_{r\times s} & \mathbf{0}_{r\times r} \end{bmatrix} = \hat{\mathbf{d}}_s(\boldsymbol{\theta})\hat{\mathbf{d}}_s^T(\boldsymbol{\theta}) \end{aligned} \tag{2.2}$$

Using this performance index in a reachability optimal control problem generates trajecto-

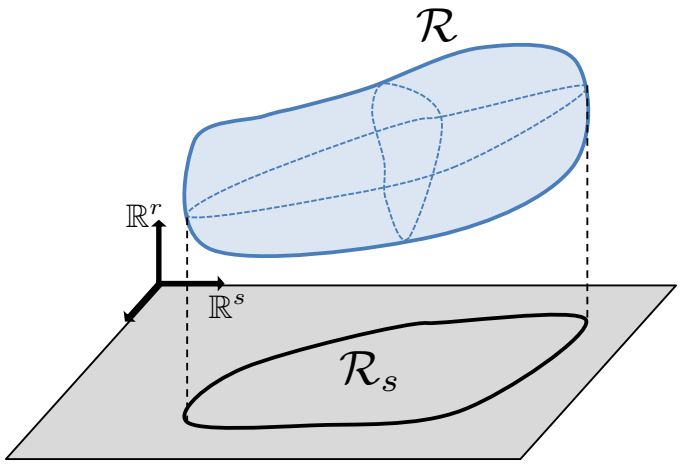


Figure 2.1: Subspace Reachability Visualization where $\mathcal{R} \subset \mathbb{R}^n$ denotes the full-state reachable volume and $\mathcal{R}_s \subset \mathbb{R}^s$ denotes the subspace reachable volume.

ries that optimize extent along the search direction parametrized by $\boldsymbol{\theta}$. Thus, by varying $\boldsymbol{\theta}$ to efficiently sample points on a hypersphere in $\mathbb{R}^s$, point solutions that lie on the subspace reachable set boundary are computed by solving the optimal control problems.

It should be noted that these subspace reachable sets are projections of the full-state reachable sets onto the coordinate axes of the states of interest. Furthermore, because the performance index in Eq. (2.2) is the squared distance along $\hat{\mathbf{d}}_s(\boldsymbol{\theta})$, the point solutions corresponding to Eq. (2.2) yield supporting hyperplanes with normal vector $\hat{\mathbf{d}}_s(\boldsymbol{\theta})$ [43]. Consequently, in this methodology the reachable set projections are convex hulls of the true subspace reachable set. This is analogous to using support functions to describe convex sets or reformulating the HJB PDE to account for unmodeled dimensions as additional disturbances [10, 16]. This class of methods for computing projections of reachability volumes are generally called projection methods.

This reachability optimal control problem is then

$$\begin{aligned} &\max_{\mathbf{u}\in U} \frac{1}{2}\mathbf{x}_f^T G(\boldsymbol{\theta})\mathbf{x}_f \\ s.t. \quad &\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \\ &\mathbf{g}(\mathbf{x}_0, t_0) = \mathbf{0} \end{aligned} \tag{2.3}$$

From optimal control theory, the first order necessary conditions of optimality can be derived as [72]

$$\begin{aligned}
\dot{\mathbf{x}} &= \frac{\partial \mathcal{H}^{*}{}^{T}}{\partial \mathbf{p}} \\
\dot{\mathbf{p}} &= -\frac{\partial \mathcal{H}^{*}{}^{T}}{\partial \mathbf{x}} \\
\mathbf{u}^{*} &= \underset{\mathbf{u}\in U}{\text{argmax}}\{\mathcal{H}\} \\
\mathcal{H}^{*} &= \mathbf{p}^{T}\mathbf{f}(\mathbf{x}, \mathbf{u}^{*}, t) \\
\mathbf{p}_0 &= -\frac{\partial g}{\partial \mathbf{x}_0}^{T}(\mathbf{x}_0)\lambda \\
\mathbf{p}_f &= \frac{\partial V}{\partial \mathbf{x}_f}^{T}(\mathbf{x}_f) = G(\boldsymbol{\theta})\mathbf{x}_f
\end{aligned} \tag{2.4}$$

where $\lambda \in \mathbb{R}$ is the Lagrange multiplier corresponding to the initial condition constraint, $g(\mathbf{x}_0)$. To solve the reachability optimal control problem described in Eq. (2.3), trajectories for the state and costate must be found to simultaneously satisfy the first order necessary conditions of optimality given in Eq. (2.4). Because the state dynamics are assumed to be at least Lipschitz continuous, the state and costate trajectories are uniquely determined by their boundary condition groups $(\mathbf{x}_0, \mathbf{p}_0, t_0)$ or $(\mathbf{x}_f, \mathbf{p}_f, t_f)$. However, due to the C^1 initial condtion set, $\mathbf{x}_0$ and $\mathbf{p}_0$ are related by the transversality conditions through the Lagrange multiplier λ. This reduces the optimal control problem solution to finding $(\mathbf{x}_0, \lambda)$ that satisfy Eq. (2.4).

2.2 Unit Ball Constraints

2.2.1 Norm Definitions and Differentiability Conditions

In previous reachability approaches using continuation methods [19, 31], the initial condition set was specified using ellipsoids such as

$$g(\mathbf{x}_0, t_0) = \mathbf{x}_0^T E \mathbf{x}_0 \leq 1 \tag{2.5}$$

where $E \in \mathbb{S}^{+}_{n\times n} > 0$, is a symmetric positive definite shape matrix that defines the ellipsoid and $\mathbf{x}_0 = \mathbf{x}(t_0)$ is the initial state. Ellipsoidal sets provide continuous differentiability of the value function with respect to the state at the initial time, ensuring the initial optimal costate to be always defined. Additionally, ellipsoidal constraints are useful because they can represent the set of states that exist within a level set of a Gaussian probability density function. These probability ellipsoids can be generated from the error covariance output of typical estimation algorithms such as minimum variance estimators or batch filters.

While many applications are well served by ellipsoidal initial constraints, others are not. Box constraints of the form $x_i \in X_i \subset \mathbb{R}, X_i = [x_{i,min}, x_{i,max}], i = 1, ..., n$ are an important case. Geometrically, these constraints describe an n-dimensional rectangle. However, as described in Section 2.3, reachability continuation methods must have constraint boundaries where the first and second derivatives ($\frac{\partial g}{\partial \mathbf{x}_0}, \frac{\partial^2 g}{\partial \mathbf{x}_0^2}$) are well-defined along the continuation path. This is equivalent to enforcing a smooth constraint boundary with no gaps or corners.

Smooth hypercube approximations can be generated by using the unit ball of the p-normed vector space. The p-norm is defined as

$$\|\mathbf{x}\|_p = \left[\sum_{i=1}^{n} |x_i|^p \right]^{\frac{1}{p}} \tag{2.6}$$

where $p \geq 1$ for this to satisfy the triangle inequality and the definition of a norm. This norm is widely used as it generalizes commonly used norms such as Manhattan/taxicab norm when $p = 1$, maximum/infinity norm when $p = \infty$, and Euclidean when $p = 2$. Under this norm and the normed vector space denoted by $(\mathbb{R}^n, \|\mathbf{x}\|_p)$, a constraint region may be specified as the closed unit ball defined as $\{\mathbf{x} \in \mathbb{R}^n : \|x\|_p \leq 1\}$. When $p = 2$, this corresponds to a unit hypersphere in $\mathbb{R}^n$. As p is increased, the unit hypersphere continuously deforms to a unit hypercube as $p \to \infty$.

It's then possible to define the p-norm unit ball initial condition set as

$$g(\mathbf{x}_0) = \|\mathbf{x}_0\|_p - 1 \leq 0 \tag{2.7}$$

which is parametrized by the scalar parameter p where $1 \leq p < \infty$. In order to use this unit ball as a constraint region in this work, the first and second derivatives should be well-defined for the continuation methods. Evaluating the first derivative of the p-norm leads to

$$\frac{\partial \|\mathbf{x}\|_p}{\partial \mathbf{x}} = \frac{\mathbf{x} \circ |\mathbf{x}|^{p-2}}{\|\mathbf{x}\|_p^{p-1}} = \frac{\text{sign}(\mathbf{x}) \circ |\mathbf{x}|^{p-1}}{\|\mathbf{x}\|_p^{p-1}} \tag{2.8}$$

where $\circ$ denotes the Hadamard/element-wise product, $|\cdot|$ denotes the element-wise absolute value, and $\text{sign}(\cdot)$ denotes the element-wise sign/signum operation.

For the case of $p = 1$, the gradient is not well-defined whenever a component of $\mathbf{x}$ is zero. This corresponds to a corner in the initial condition set where there are multiple tangent hyperplanes to the set at these points. One feasible choice for this scenario is to define $|0|^0 = 1$. In this case, the ith component of the gradient is zero whenever ith component of the state is zero. This ensures continuity in the limit sense with respect to the state.

Additionally, for any value of $p \geq 1$, the gradient is not defined at the origin. In this chapter, singleton initial condition sets are not considered. As a result, this work focuses on initial condition set regions which have non-zero volume.

The second derivative of the p-norm leads to

$$\frac{\partial}{\partial x_j}\left[\frac{\partial \|\mathbf{x}\|_p}{\partial x_i}\right] = (p-1)\frac{\partial x_i}{\partial x_j}\frac{|x_i|^{p-2}}{\|\mathbf{x}\|_p^{p-1}} + (1-p)\frac{x_i|x_i|^{p-2}x_j|x_j|^{p-2}}{\|\mathbf{x}\|_p^{2p-1}} \tag{2.9}$$

where $i, j = 1, 2, \ldots, n$ denote the first and second indices of the resultant matrix. Note the $\frac{\partial x_i}{\partial x_j}$ term is equivalent to the Kronecker delta δ_{ij} used commonly in index and Einstein summation notation. In matrix form, $\frac{\partial x_i}{\partial x_j}$, represents the $n \times n$ identity matrix, $\mathbb{I}_{n\times n}$. Also

note the second derivative is not well-defined whenever $p < 2$ and any component of $\mathbf{x}$ is zero.

These results mirror the differentiabiltiy conditions outlined by Sundaresan [74]. Theorem 8 from this paper proves that if $1 \leq p < \infty$, p-norms are of class C^{∞} if p is an even integer, class C^{p-1} if p is an odd integer, and of class $C^{E(p)}$ if p is not an integer [74]. $E(p)$ denotes the integral part of the positive real number p, equivalent to the floor(p). As aforementioned, the numerical continuation methods require a constraint that is at least C^2. This results in a requirement for exact numerical continuation that $p \geq 2$.

As discussed, the p-norm constraint region has differentiability issues near the origin. An alternative norm may be used to avoid these issues. By removing the exponent $1/p$ from the p-norm, an F-norm is achieved [30].

$$\|\mathbf{x}\|_F = \sum_{i=1}^{n} |x_i|^p \tag{2.10}$$

The L^p spaces are F-spaces for all $p \geq 0$. Please note the the F-norm does not denote the Frobenius norm but instead it represents the corresponding metric for an F-space. The closed unit ball for the vector space $(\mathbb{R}^n, \|\mathbf{x}\|_p)$ and for the vector space $(\mathbb{R}^n, \|\mathbf{x}\|_F)$ are identical where $\|\mathbf{x}\|_F$ is defined in Eq.(2.10). Consequently, the p-norm unit ball set may be replaced with the defined F-norm along with its first and second derivatives. Evaluating both of these derivatives leads to

$$\frac{\partial \|\mathbf{x}\|_F}{\partial \mathbf{x}} = p\, \mathbf{x} \circ |\mathbf{x}|^{p-2} = p\, \text{sign}(\mathbf{x}) \circ |\mathbf{x}|^{p-1} \tag{2.11}$$

$$\frac{\partial^2 \|\mathbf{x}\|_F}{\partial \mathbf{x}^2} = p(p-1)\, \text{diag}(|\mathbf{x}|^{p-2}) \tag{2.12}$$

where diag$(\cdot)$ represents a diagonal matrix with main diagonal given as the argument.

Similarly to the p-norm, the F-norm shares the same differentiability conditions as a function of p. However, the derivatives are simpler to compute as there is no coupling between components of the state in the F-norm. As a result, the ith component of the state only affects derivatives with respect to that component. Furthermore, the F-norm has well-defined derivatives at the origin. The following is another equivalent expression for the initial condition set using the unit ball by F-norm as

$$g(\mathbf{x}_0) = \|\mathbf{x}_0\|_F - 1 \leq 0 \tag{2.13}$$

which is parametrized by the scalar parameter p where $1 \leq p < \infty$.

To use numerical continuation for cases where $p < 2$, approximate methods must be used. A simple method for closely approximating the first and second derivatives of the defined norms are to use the relationship $|x| = \sqrt{x^2}$. An approximate absolute value function can then be defined as

$$|x| \approx |x|_s = \sqrt{x^2 + \epsilon^2} \tag{2.14}$$

where $\epsilon \in \mathbb{R}$ is a smoothing parameter that defines the accuracy of the approximation. As $\epsilon \to 0$, $|x|_s \to |x|$ and the accuracy of the approximation is improved. This smoothed absolute value function ensures $|x|^y$ is well-defined when $y < 0$ and $x = 0$.

2.2.2 Affine Transformations of Unit Ball

The initial condition sets defined in Eq.(2.7) and Eq.(2.13) result in the set of states contained within the unity level set of the norm, resulting in unit circles, unit spheres, unit cubes, etc defined relative to the origin. However, it is possible to scale, rotate, and translate these initial condition sets using an affine transformation. If $M \in \mathbb{R}^{n \times n}$ is the invertible transformation matrix such that $\tilde{\mathbf{x}} = M(\mathbf{x} - \mathbf{x}_c)$, $\mathbf{x}_c$ defines the center of the constraint re-

gion, and $\|\tilde{\mathbf{x}}\| \leq 1$, the initial condition sets and derivatives can be evaluated as

$$g(\mathbf{x}_0) = \|\tilde{\mathbf{x}}_0\| - 1 \leq 0 \tag{2.15}$$

$$\frac{\partial g}{\partial \mathbf{x}}(\mathbf{x}_0) = M^T \frac{\partial \|\tilde{\mathbf{x}}_0\|}{\partial \tilde{\mathbf{x}}} \tag{2.16}$$

$$\frac{\partial^2 g}{\partial \mathbf{x}^2}(\mathbf{x}_0) = M^T \frac{\partial^2 \|\tilde{\mathbf{x}}_0\|}{\partial \tilde{\mathbf{x}}^2} M \tag{2.17}$$

where $\tilde{\mathbf{x}}_0 = M(\mathbf{x}_0 - \mathbf{x}_c)$. This allows for the representation of scaled, rotated, and translated sets such as n-dimensional rectangles and ellipsoids. This type of transformation may be applied to the unit ball of any normed vector space, including the p-norm and F-norm previously discussed.

2.2.3 Feasible Control Set

In a similar way, the set of feasible control inputs in previous work is prescribed using spherical sets such as

$$U = \{\mathbf{u} : \mathbf{u}^T \mathbf{u} \leq u_m^2\} \tag{2.18}$$

where $u_m \in \mathbb{R}^+$ defines the maximum control input Euclidean norm. Ellipsoidal control input constraints were also represented by adjusting the system dynamics to include a positive definite transformation matrix on the control input term.

The set of feasible controls may also be defined using the unit balls of the vector spaces such as $(\mathbb{R}^m, \|\mathbf{u}\|_p)$ and $(\mathbb{R}^m, \|\mathbf{u}\|_F)$. This allows for non-ellipsoidal feasible control sets in the reachability analysis.

Moreover, the feasible control set does not need to be centered around the origin or contain the origin at all. This allows for minimum-time reachability analyses in which feasible

control set defines the allowable deviations from a nominal control input signal, $\mathbf{u}_c(t)$ or $\mathbf{u}_c(\mathbf{x}, t)$.

2.2.4 Maximal Inner Product on Unit Ball

For constraint sets on both states and control input, the problem of maximizing inner products over the unit balls arises. The optimization problem is stated as

$$\begin{aligned} &\max_{\mathbf{x}} \ \mathbf{y}^T\mathbf{x} \\ &s.t. \ \|\mathbf{x}\|_p \leq 1 \end{aligned} \tag{2.19}$$

The solution relies on the dual norm defined on a normed vector space $(\mathbb{R}^n, \|\mathbf{x}\|)$

$$\|\mathbf{y}\|_* := \sup\{\mathbf{y}^T\mathbf{x} : \mathbf{x} \in \mathbb{R}^n, \|\mathbf{x}\| \leq 1\} \tag{2.20}$$

which shows that maximal value of $\mathbf{y}^T\mathbf{x}$ is $\|\mathbf{y}\|_*$. For the p-norm, Hölder's inequality shows that the dual is the q-norm where $\frac{1}{p} + \frac{1}{q} = 1$ [30]. So an equivalent problem for Eq. (2.19) is to find the $\mathbf{x}$ such that $\mathbf{y}^T\mathbf{x} = \|\mathbf{y}\|_q$ where $q = \frac{p}{p-1}$.

This is accomplished by choosing

$$\begin{aligned} \mathbf{x}^* &= \frac{\mathbf{v}}{\|\mathbf{v}\|_p} \\ \mathbf{v} &= \text{sign}(\mathbf{y}) \circ |\mathbf{y}|^{q-1} \end{aligned} \tag{2.21}$$

Proof: Using this choice in $\mathbf{x}^*$,

$$\|\mathbf{x}^*\|_p = \left[\sum_{i=1}^{n} \left|\frac{v_i}{\|\mathbf{v}\|_p}\right|^p\right]^{\frac{1}{p}} = \frac{1}{\|\mathbf{v}\|_p}\left[\sum_{i=1}^{n} |v_i|^p\right]^{\frac{1}{p}} = 1 \tag{2.22}$$

which shows that $\mathbf{x}^*$ lies on the boundary of the unit ball for the $(\mathbb{R}^m, \|\mathbf{u}\|_p)$ vector space.

Furthermore,

$$\begin{aligned} \|\mathbf{v}\|_p^p = \sum_{i=1}^n |v_i|^p = \sum_{i=1}^n |\text{sign}(y_i)|y_i|^{q-1}|^p = \sum_{i=1}^n ||y_i|^{q-1}|^p = \sum_{i=1}^n |y_i|^q = \|\mathbf{y}\|_q^q \\ \mathbf{y}^T\mathbf{v} = \sum_{i=1}^n y_i v_i = \sum_{i=1}^n y_i \text{sign}(y_i)|y_i|^{q-1} = \sum_{i=1}^n |y_i|^q = \|\mathbf{y}\|_q^q \end{aligned} \tag{2.23}$$

where the relationship between q and p is used. Evaluating the inner product between $\mathbf{y}$ and $\mathbf{x}^*$ and using the results from Eq. (2.23),

$$\mathbf{y}^T\mathbf{x}^* = \sum_{i=1}^n y_i x_i^* = \frac{1}{\|\mathbf{v}\|_p} \sum_{i=1}^n y_i v_i = \frac{1}{\|\mathbf{y}\|_q^{\frac{q}{p}}} \|\mathbf{y}\|_q^q = \|\mathbf{y}\|_q^{q-\frac{q}{p}} = \|\mathbf{y}\|_q \tag{2.24}$$

□

A corollary result comes from the form of $\mathbf{x}^*$.

$$\mathbf{x}^* = \frac{\text{sign}(\mathbf{y}) \circ |\mathbf{y}|^{q-1}}{\|\mathbf{v}\|_p} = \frac{\text{sign}(\mathbf{y}) \circ |\mathbf{y}|^{q-1}}{\|\mathbf{y}\|_q^{q-1}} = \frac{\partial \|\mathbf{y}\|_q}{\partial \mathbf{y}} \tag{2.25}$$

which shows that the optimizing vector to Eq. (2.19) can be evaluated using the first derivative of the q-norm. Geometrically, $\mathbf{x}^*$ is a normal vector to the $\|\mathbf{y}\|_q$ level set evaluated at $\mathbf{y}$. It should be noted that $\mathbf{x}^*$ shares the same existence conditions as the conditions for C^1 q-norm. Consequently when $\mathbf{y} = 0$, $p = 1$, or $p = \infty$, there is not a unique solution to the maximal inner product problem.

The Lagrange multiplier corresponding to the optimization problem in Eq. (2.19) can be found using the first order necessary condition of optimality for the constrained optimization problem. In this case, the optimizing solution is known to lie on the boundary of the constraint region, so an equality constraint can be used in Eq. (2.19). The resulting necessary condition of optimality from the first variation of the optimization Lagrangian is given

by

$$\begin{aligned} 0 = y_i + \lambda \frac{x_i |x_i|^{p-2}}{\|\mathbf{x}\|_p^{p-1}} &= y_i + \lambda \frac{v_i}{\|\mathbf{v}\|_p} \left| \frac{v_i}{\|\mathbf{v}\|_p} \right|^{p-2} \\ &= y_i + \lambda \frac{1}{\|\mathbf{v}\|_p^{p-1}} v_i |v_i|^{p-2} = y_i + \lambda \frac{1}{\|\mathbf{v}\|_p^{p-1}} y_i \end{aligned} \tag{2.26}$$

where λ denotes the Lagrange multiplier. Eq. (2.26) must hold for $i = 1, 2, \ldots, n$ and arbitrary real, finite y_i. Solving for the Lagrange multiplier,

$$\lambda = -\|\mathbf{v}\|_p^{p-1} = -\|\mathbf{y}\|_q \tag{2.27}$$

As the closed unit balls for the p-norm and F-norm case are identical, it follows that the $\mathbf{x}^*$ that maximizes $\mathbf{y}^T\mathbf{x}$ subject to $\|\mathbf{x}\|_F \leq 1$ is the same as described in Eq. (2.21) and Eq. (2.25). While the solution for $\mathbf{x}^*$ is identical to the p-norm case, the corresponding Lagrange multipliers are different. Using a similar approach to Eq. (2.26), the Lagrange multiplier for the F-norm case is $\lambda = -\frac{\|\mathbf{y}\|_q}{p}$. Additionally, any derivatives of $\mathbf{x}^*$ with respect to $\mathbf{y}$ are also different depending on which norm is being used to define the unit ball.

2.3 Continuation Methods

The purpose of continuation methods is to solve a complex optimal control problem by solving a simpler one and deforming the simple solution to the complex solution. This deformation is generally performed using a scalar continuation parameter which parametrizes the optimal control problem [21]. The two-point boundary value problem described in Eq. (2.4) is equivalent to the following root-finding problem

$$\mathbf{F}[\mathbf{z}(s), s] = \mathbf{0} \tag{2.28}$$

where s is the scalar continuation parameter and

$$\mathbf{F} = \begin{bmatrix} \frac{\partial V}{\partial \mathbf{x}_f}^T(\mathbf{x}_f) - \mathbf{p}_f \\ g(\mathbf{x}_0) \end{bmatrix}$$
$$\mathbf{z} = \begin{bmatrix} \mathbf{x}_0 \\ \lambda \end{bmatrix} \tag{2.29}$$

Thus, as the continuation parameter s changes, the solution to the corresponding optimal control problem $\mathbf{z}$ changes. Because the necessary conditions described in $\mathbf{F}$ and the solution vector $\mathbf{z}$ depend on the continuation parameter the following differential equation must hold

$$\frac{d\mathbf{F}[\mathbf{z}(s), s]}{ds} = \mathbf{0} \tag{2.30}$$

as $\mathbf{F} = \mathbf{0}$ along the optimal solution curve $(\mathbf{z}(s), s)$. Further the differential equation in Eq. (2.30),

$$\frac{d\mathbf{F}}{ds} = \frac{\partial \mathbf{F}}{\partial \mathbf{z}}\frac{d\mathbf{z}}{ds} + \frac{\partial \mathbf{F}}{\partial s} = \mathbf{0} \tag{2.31}$$

which results in the following initial-value problem in terms of the continuation parameter

$$\frac{d\mathbf{z}}{ds} = -\left[\frac{\partial \mathbf{F}}{\partial \mathbf{z}}\right]^{-1}\frac{\partial \mathbf{F}}{\partial s} \tag{2.32}$$

Eq. (2.32) represents how the optimal control problem solution changes as the continuation parameter is varied. Therefore, if the solution of the optimal control problem $(\mathbf{z}(s_0))$ can be computed, and the jacobian $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$ is invertible throughout the entire solution curve $\mathbf{z}(s)$, integrating Eq. (2.32) to $s = s_f$ will result in the optimal control problem solution $(\mathbf{z}(s_f))$. The solution procedure to the overall optimal control problem is then reduced to an initial-value problem which can be solved accurately with a number of numerical integration methods.

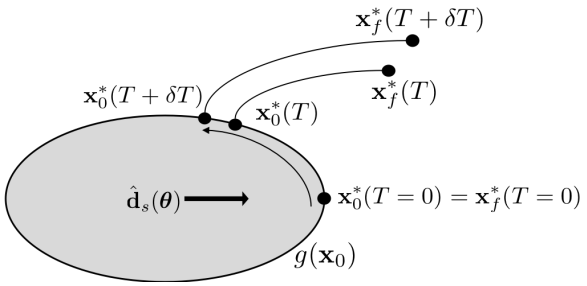


Figure 2.2: 2-dimensional continuation method illustration

For time-based reachability problems, a useful choice in the continuation parameter is in the terminal time horizon, $t_f = t_0 + T$. By using the time horizon as a continuation parameter and integrating Eq. (2.32), the reachability optimal control problem solution for Eq. (2.3) is solved for each time horizon. Consequently, the time-horizon evolution of a sample of the reachable set boundary, parameterized by $\boldsymbol{\theta}$, is computed. By performing a collection of these continuation method initial value problems, the time-horizon evolution of samples of the reachable set boundary are computed.

At the initial time horizon $T = 0$, $t_0 = t_f$, $\mathbf{x}_0 = \mathbf{x}_f$, and $\mathbf{p}_0 = \mathbf{p}_f$. Finding the optimal control problem solution is equivalent to solving the following constrained optimization problem

$$\begin{aligned} \max_{\mathbf{x}_0} \frac{1}{2}\mathbf{x}_0^T G(\boldsymbol{\theta})\mathbf{x}_0 &= \frac{1}{2}(\hat{\mathbf{d}}_s(\boldsymbol{\theta})^T\mathbf{x}_0)^2 \\ s.t. \quad g(\mathbf{x}_0) &= \|M(\mathbf{x}_0 - \mathbf{x}_c)\|_{p,F} - 1 = 0 \end{aligned} \tag{2.33}$$

where M is the linear transformation matrix to the unit ball that describes the size and shape of the initial condition set, $\mathbf{x}_c$ is the center of the initial condition set, and either p-norm or F-norm classes are used to define the normed unit ball. Note that this is a slightly modified version of the optimization problem in Eq. (2.19) where $\mathbf{y} = \hat{\mathbf{d}}_s$ and the objective is quadratic in terms of the inner product. By restricting the domain to states such that $\hat{\mathbf{d}}_s^T\mathbf{x} > 0$, the solution to this problem is the same as in Eq. (2.19). However, the Lagrange

multipliers differ in this squared inner product case.

$$
\begin{aligned}
q &= \frac{p}{p-1} \\
\tilde{\mathbf{x}}_0^* &= \frac{\text{sign}(\hat{\mathbf{d}}_s) \circ |\hat{\mathbf{d}}_s|^{q-1}}{\|\hat{\mathbf{d}}_s\|_q^{q-1}} \\
\mathbf{x}_0^* &= M^{-1}\tilde{\mathbf{x}}_0^* + \mathbf{x}_c \\
\lambda &= \begin{cases} -\|\hat{\mathbf{d}}_s\|_q^2 & \text{if p-norm used} \\ -\frac{1}{p}\|\hat{\mathbf{d}}_s\|_q^2 & \text{if F-norm used} \end{cases}
\end{aligned}
\tag{2.34}
$$

Once this initial value of $\mathbf{z}(T=0)$ is computed, a continuation method integration shown in Eq. (2.32) can be performed to get the optimal control solution over multiple time horizons. An illustration of this concept for a single point solution is shown in Fig. 2.2. For the reachability problem given in Eq. (2.3), Holzinger et al. has computed $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$ and $\frac{\partial \mathbf{F}}{\partial T}$ in the continuation method in terms of state transition matrices and optimal Hamiltonian dynamics [19]. Using this methodology, Holzinger et al. were able to compute 1-dimensional subspace reachable set for a 6-dimensional nonlinear duffing oscillator system in 20 seconds on a MacBook Pro 2.4GHz Intel Core 2 Duo processor with 4GB 667MHz DDR2 SDRAM [19].

Solving the differential equation described in Eq. (2.32) relies on the invertibility of the Jacobian matrix $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$ throughout the entire solution path. For many cases, this condition is met by effectively using preconditioning to accurately solve jacobians that are numerically near-singular. Furthermore, as each continuation method differential equation represents the solution path for a single particle, it is always possible to spawn a nearby sample to avoid the singularity.

However, it is possible to reparametrize the solution path in terms of arclength instead of reachability time horizon, T, to alleviate some of the singularity issues. If σ denotes an

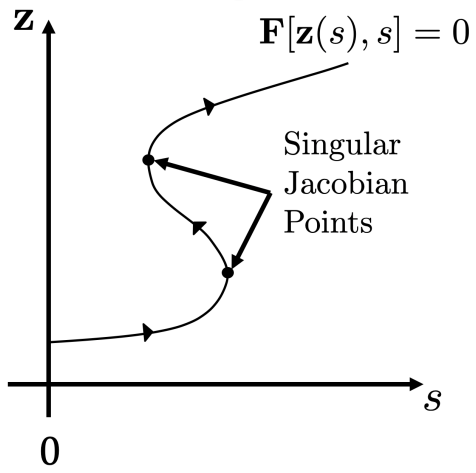


Figure 2.3: Pseudo-arclength Continuation Visualization

arclength parameter,

$$\begin{aligned} \frac{d\mathbf{F}[\mathbf{z}(s(\sigma)), s(\sigma)]}{d\sigma} = \frac{\partial \mathbf{F}}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \sigma} + \frac{\partial \mathbf{F}}{\partial T}\frac{\partial T}{\partial \sigma} &= \begin{bmatrix} \frac{\partial \mathbf{F}}{\partial \mathbf{z}} & \frac{\partial \mathbf{F}}{\partial T} \end{bmatrix} \begin{bmatrix} \frac{\partial \mathbf{z}}{\partial \sigma} \\ \frac{\partial T}{\partial \sigma} \end{bmatrix} \\ &= A\vartheta \\ &= \mathbf{0} \end{aligned} \tag{2.35}$$

where $\|\vartheta\|_2 = 1$ [20]. In this version of continuation, the null space of the augmented Jacobian, A, must be computed where $A \in \mathbb{R}^{n+1 \times n+2}$. The null space of A is one-dimensional iff either

1. $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$ is non-singular or
2. dim[Null($\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$)] = 1 and $\frac{\partial \mathbf{F}}{\partial T} \notin$ Range($\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$)

where Null($\cdot$) denotes the null space or kernel and Range($\cdot$) denotes the range or image.

For the usual case where the dim[Null(A)] = 1, there are exactly two unit vectors ϑ that satisfy Eq. (2.35) corresponding to the two directions to travel along the solution curve. In general, the initial value for ϑ should be chosen such that $\frac{\partial T}{\partial \sigma}$ is the desired sign. From there, one can use continuity to ensure the next iteration of ϑ has a positive inner product with

respect to its previous iteration. Alternatively, one can define another augmented matrix

$$\bar{A} = \begin{bmatrix} A \\ \vartheta^T \end{bmatrix} \tag{2.36}$$

and determine which ϑ to use based on maintaining a consistent sign of det($\bar{A}$).

Eq. (2.35) with the initial condition of $\mathbf{z}(\sigma = 0) = \mathbf{z}(T = 0)$ and $T(\sigma = 0) = 0$ defines an initial value problem that may be numerically integrated from $\sigma = 0$ until the condition $T = T_f$ is satisfied.

Singular points in $\mathbb{R}^{n+1}$ are points in which the dim[Null(A)] ¿ 1. These singular points are commonly used to signal bifurcation points in the solution curve in which two or more curves branch from a solution path. In the framework of the sample-based reachability optimal control control problem in Eq. (2.3), these bifurcation points could signify the development of concave regions of the reachable volume. While there are numerical techniques for detecting these bifurcation points [20], these techniques are outside of the scope of this thesis.

2.4 Backwards Reachability Methodology

It is convenient to define trajectories using flow functions as follows

$$\begin{aligned} \mathbf{x}(t) &= \boldsymbol{\phi}_x(t; \mathbf{x}_0, \mathbf{p}_0, t_0) \\ \mathbf{p}(t) &= \boldsymbol{\phi}_p(t; \mathbf{x}_0, \mathbf{p}_0, t_0) \end{aligned} \tag{2.37}$$

where the semicolon is used to separate the independent parameter of time from the boundary conditions that uniquely identify the trajectory based on the flow equations in Eq. (2.4).The

forward reachable set for a specified time horizon is then defined as

$$\mathcal{R}(t_f; U, \mathbf{f}, g, t_0) = \{\mathbf{x}_f \in \mathbb{R}^n : \forall \mathbf{u}(t) \in U, g(\mathbf{x}_0) \leq 0, \mathbf{x}_f = \boldsymbol{\phi}_x(t_f; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \lambda), t_0), t \in [t_0, t_f]\} \tag{2.38}$$

Intuitively, the forward reachable set defines all the states in state space that can be achieved by the state flow function starting from some initial condition set using admissible control at the specified time, $t_f > t_0$. A point in state space is forwards reachable at time t_f if it belongs to the forwards reachable set for that time horizon, $\mathcal{R}(t_f; U, \mathbf{f}, g, t_0)$.

Backwards reachable sets can be defined in a similar fashion as Eq. (2.38)

$$\mathcal{R}^-(\tau_f; U, \mathbf{f}, g, \tau_0) = \{\mathbf{x}_0 \in \mathbb{R}^n : \forall \mathbf{u}(\tau) \in U, g(\mathbf{x}_f) \leq 0, \mathbf{x}_0 = \tilde{\boldsymbol{\phi}}_x(\tau_f; \mathbf{x}_f, \mathbf{p}_f(\mathbf{x}_f, \lambda), \tau_0), \tau \in [\tau_0, \tau_f]\} \tag{2.39}$$

where $\tau = -t$ is defined for backwards reachable sets as a nonnegative and increasing time parameter and $\tilde{\boldsymbol{\phi}}_x$ is a modified flow function from Eq. (2.37) in which time flows in reverse. Said differently, the backwards reachable set defines all the initial states that can achieve the terminal condition set using admissible control at the specified time horizon τ. Likewise, a point in state space is backwards reachable at time τ if it belongs to the backwards reachable set for that time horizon, $\mathcal{R}^-(\tau_f; U, \mathbf{f}, g, \tau_0)$.

The aforementioned methodology can be used to compute backwards reachable sets with a few minor changes. The modified optimal control problem for the backwards reachability case can be expressed as

$$\begin{aligned} \max_{\mathbf{u} \in U} & \frac{1}{2}\mathbf{x}(\tau = \tau_f)^T G(\boldsymbol{\theta})\mathbf{x}(\tau = \tau_f) \\ s.t.\ \ \tfrac{d\mathbf{x}}{d\tau} = \overset{\circ}{\mathbf{x}} &= \tilde{\mathbf{f}}(\mathbf{x}, \mathbf{u}, \tau) = -\mathbf{f}(\mathbf{x}, \mathbf{u}, -\tau) \\ & g(\mathbf{x}(\tau = \tau_0)) = 0 \end{aligned} \tag{2.40}$$

Furthermore, the optimal control Hamiltonian and resulting optimal control policy is mod-

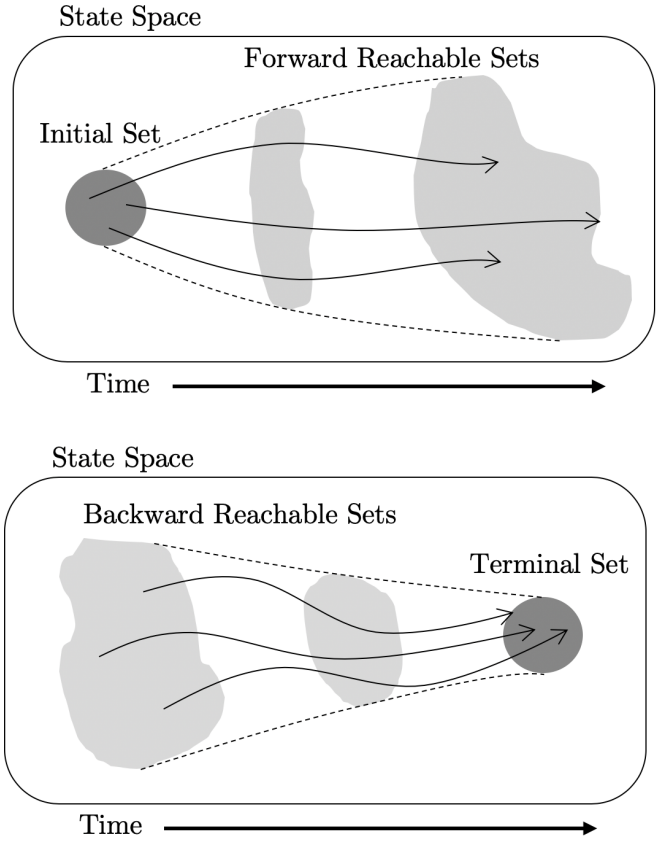


Figure 2.4: Illustration highlighting differences between forward and backward reachability. The darker circular regions denote the user-specified boundary condition.

ified as

$$\mathcal{H}^* = \max_{\mathbf{u}\in U}\{\mathbf{p}^T\tilde{\mathbf{f}}(\mathbf{x},\mathbf{u},\tau)\} = \min_{\mathbf{u}\in U}\{\mathbf{p}^T\mathbf{f}(\mathbf{x},\mathbf{u},-\tau)\} \tag{2.41}$$

Because the boundary conditions on the state and costate variables are independent of time, the continuation method outlined in Sec. 2.3 can be used to find the backwards reachability optimal control problem solution from $\tau = 0$ to $\tau = T > 0$.

For many reachability problems, it is useful to know whether or not a state was reachable at any time throughout the given time horizon. This differs from the previous reachable set definitions that identify states that are reachable exactly at the specified time horizon. This definition of the reachable set allows for states to be reachable at certain times and unreachable at other times. This implies that this system is not small-time local controllable (STLC). A system is STLC if for every state, the system can remain near that state for all times and can reach nearby states in arbitrarily small amounts of time [75]. Intuitively, this means there is no admissible control possible to cancel the system dynamics.

However, many important types of real-world dynamical systems are not STLC, resulting in different interpretations of what it means for a state to be reachable. For example, in safety analysis problems, it is useful to know whether or not a state was ever unsafe throughout the given time horizon. In situations like these, once a state is labelled unsafe (reachable), then that state cannot become safe (unreachable) at some later time. Visually, this means the reachable set size never decreases as the time horizon increases. This modified reachable set is often called the reachable tube.

The reachability optimal control Hamiltonian, defined as the inner product between the costate and the optimal state dynamics, gives a measure of the local expansion or contraction of the reachable set. Therefore, to ensure the size of the reachable set never decreases in this forward reachability case, the optimal control Hamiltonian must be constrained so it is always nonnegative. This is a similar approach performed by Lygeros, Tomlin, and

Mitchell in computations of backwards reachable tubes using differential games [5, 1]. This modifies the classical forward reachability HJB PDE to

$$\begin{aligned} \frac{\partial V}{\partial t} + \min_{\mathbf{u}\in U}\Big[0, \mathcal{H}(\mathbf{x}, \mathbf{u}, \mathbf{p}, t)\Big] = 0 \\ V(\mathbf{x}, t_0) = g(\mathbf{x}) \end{aligned} \tag{2.42}$$

and the corresponding forward reachable tube definition

$$\mathcal{R}_t(t_f; U, \mathbf{f}, g, t_0) = \{\mathbf{x}_f \in \mathbb{R}^n : \forall \mathbf{u}(t) \in U, g(\mathbf{x}_0) \leq 0, \mathbf{x}_f = \boldsymbol{\phi}_x(t; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \lambda), t_0), \forall t \in [t_0, t_f]\} \tag{2.43}$$

Similarly, the modified backwards reachability HJB PDE is given by

$$\begin{aligned} \frac{\partial V}{\partial t} + \min_{\mathbf{u}\in U}\Big[0, \mathcal{H}(\mathbf{x}, \mathbf{u}, \mathbf{p}, t)\Big] = 0 \\ V(\mathbf{x}, t_0) = g(\mathbf{x}) \end{aligned} \tag{2.44}$$

and the corresponding backwards reachable tube definition

$$\mathcal{R}_t^-(\tau_f; U, \mathbf{f}, g, \tau_0) = \{\mathbf{x}_0 \in \mathbb{R}^n : \forall \mathbf{u}(\tau) \in U, g(\mathbf{x}_f) \leq 0, \mathbf{x}_0 = \tilde{\boldsymbol{\phi}}_x(\tau; \mathbf{x}_f, \mathbf{p}_f(\mathbf{x}_f, \lambda), \tau_0), \forall \tau \in [\tau_0, \tau_f]\} \tag{2.45}$$

Intuitively, this reachable tube is the union of the the reachable sets up to the particular time horizon as shown in Eq. (2.46) for the forward case. Likewise, the backwards reachable tube defined in Eq. (2.45) is the union over time of the backwards reachable set defined in Eq. (2.39)

$$\mathcal{R}_t(t_f; U, \mathbf{f}, g, t_0) = \bigcup_{\hat{t}\in[t_0, t_f]} \mathcal{R}(\hat{t}; U, \mathbf{f}, g, t_0) \tag{2.46}$$

If a system is indeed STLC everywhere, then the reachable set and reachable tube are identical.

As the reachable sets are computed using sampling methods, minor modifications need to

be made to compute reachable tubes as opposed to reachable sets. To enforce the sign constraints on the optimal control Hamiltonian, the point solutions are frozen whenever the optimal control Hamiltonian is the incorrect sign. This prevents point solutions from decreasing the size of the reachable tube boundary. To freeze a point solution, one must set the time horizon derivative to zero ($d\mathbf{z}/dT = 0$). This is analogous to the freezing approach performed by Mitchell et al. in the computation of backwards reachable tubes using viscosity solutions to the modified HJB PDE in Eq. (2.44).

Once a point solution is frozen, it can unfreeze to resume the nominal continuation method integration. For a point solution to unfreeze, it must simultaneously meet two criteria

1. Optimal Control Hamiltonian ($\mathcal{H}^*$) is the correct sign
2. Performance index V must exceed or match the value at which the point solution was frozen

The first of the criteria ensures the point solution is increasing the size of the reachable tube while the second criteria ensures the point solution is an extremal point and lies on the boundary of the reachable tube. To unfreeze a point solution, allow the continuation method to resume using Eq. (2.32). However, the continuation method integration should not begin from the frozen point solution but rather from the point solution from the undisturbed continuation method integrated point solution at that time horizon. This is because the continuation method already computes the extremal point solutions at a given time horizon. Algorithmically, the computation of the reachable tubes can be performed as a post-processing step to the computation of the reachable sets. In this way, using this approach both reachable tubes and reachable sets are computed simulataneously.

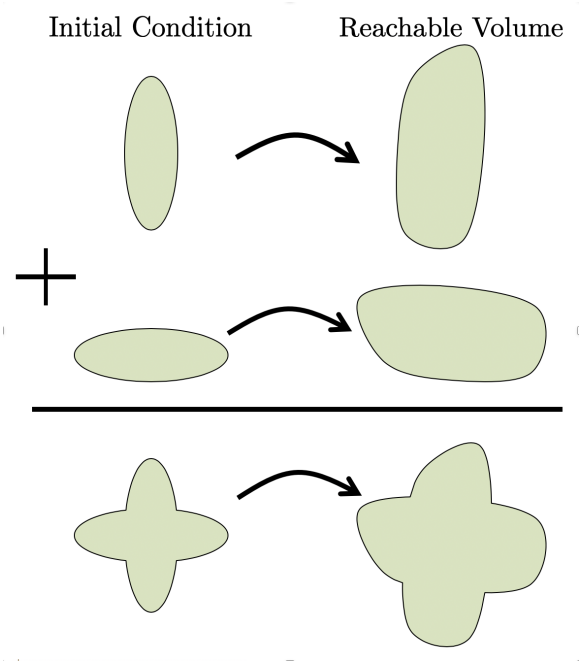


Figure 2.5: Illustration of union of independent initial condition sets with union of resulting reachable volumes

2.5 Unions of Reachable Volumes

As previously discussed, reachability continuation methods must have initial condition sets that are at least C^2 everywhere along the boundary of the set. This suggests that a piecewise definition of the initial condition set are appropriate for this type of analysis. However, piecewise definitions of the initial condition set may be challenging for certain scenarios.

Another technique for solving reachability problems with piecewise defined initial condition sets is to use represent it using the union of multiple, independent initial condition sets. Intuitively, the union of the individual independent initial condition sets results in the union of the resulting reachable volumes. If each of the constituent sets are convex and C^2, separate sample-based reachability analyses can be performed then combined to achieve the overall reachable volume. This concept is visualized in Fig. 2.5.

This allows for reachability analyses on non-convex initial condition sets. Furthermore, there are no guarantees that the union of the constituent reachable volumes are convex as

the time horizon increases.

This concept can also be used to approximate a large class of initial condition sets using the union of multiple convex and C^2 sets. For example, one can approximate a square initial condition set with a collection of ellipsoids or circles. While the normed unit ball approach may be used instead to yield an approximation to the square initial set, the numerical conditioning of the continuation method will be improved when using ellipsoids or circles to approximate the set.

Numerically, one has to take care in reducing the number of redundant search directions and reachability point solutions generated by this technique. This is because with every independent initial condition set, a separate reachability analysis is performed. Using the sample-based continuation reachability approach described in this chapter, one can restrict the reachable set samples and search directions to be outward facing to help ensure redundant reachability samples are not created.

2.6 Accuracy Considerations

When computing the numerical solutions to a continuation method problem, one can quantify an error based on a distance metric from the solution path. In the reachability formulation described in this chapter, the solution curves correspond to satisfaction of the first order necessary conditions of optimality given in Eq. (2.29). By ensuring the first order necessary conditions of optimality are satisfied in the reachability problem, local optimal solutions are guaranteed. Furthermore, the amount of error in the optimal solution is proportional to $\|\mathbf{F}(\mathbf{z})\|$.

By maintaining $\|\mathbf{F}(\mathbf{z})\| \leq F_{\text{thres}}$ for every sample of the reachable volume, the overall numerical error in the reachable volume calculation is bounded. This can be assured during the numerical continuation method in two ways. The first is by performing accurate

numerical integration of the continuation method differential equations. Many numerical differential equation solvers have user-specified error tolerances which tradeoff increased accuracy with faster computations. The second way in which the reachability accuracy is assured is with corrective Newton's or Broyden's-type steps whenever the solution error $\|\mathbf{F}(\mathbf{z})\|$ exceeds a user-specified threshold F_{thres}.

These corrective, error-reducing steps may also be quickly performed after the numerical continuation is complete. For the scenarios in which only the terminal time reachable volume is desired, this may lead to large computational improvements. In these cases, the user may set "loose" numerical tolerances throughout the numerical integration to avoid too many intermediate function calls. Then after the numerical integration is performed, a Newton's or Broyden's refinement process is then completed to reduce the overall error at the desired reachability time horizon.

2.7 Numerical Considerations

There are two primary sources of error with the continuation methods outlined in Sec. 2.3: error due to numerical integration and error due to numerical conditioning of the linearized system in Eq. (2.32).

As the continuation method integration time horizon T increases, the point solutions exhibit increasingly larger residuals when satisfying the first order necessary conditions of optimality in Eq. (2.29). In this work, an adaptive Runge-Kutta-Fehlberg numerical scheme is used to integrate the state, costate, and state transition matrix flows as well as the optimal control problem solution trajectories in the continuation method [76].

While this implementation is commonly used in practice and straightforward to implement, Runge-Kutta schemes are not symplectic. Symplectic integrators preserve the structure of Hamiltonian systems and thus very nearly preserve conserved quantities such as

total system energy. Using the transversality conditions in optimal control and first order necessary conditions, one can show that the optimal control Hamiltonian is constant for time-independent systems [72]. Therefore, symplectic integration schemes have benefits for long-duration reachability problems. For the demonstration of the proposed methodologies of this thesis however, the embedded Runge-Kutta integration scheme is sufficient at maintaining accuracy and satisfying the first order necessary conditions of optimality over the given time horizons.

The continuation method solution dynamics shown in Eq. (2.32) is given by the solution of a linear system of the form $M\mathbf{y} = \mathbf{b}$ where M is the Jacobian matrix of the optimality constraints with respect to the optimal control problem solution. Depending on the constraint dynamics and scale between the coordinates of the optimal control problem solution $\mathbf{z}$, as T increases the numerical condition number of the M may increase arbitrarily. This increase in condition number is a well-known problem often seen in numerical shooting or multiple-shooting methodologies [77]. In numerical linear algebra, the condition number of a matrix, denoted as $\kappa(M)$, gives a measure of the sensitivity of the solution to the general linear system $M\mathbf{y} = \mathbf{b}$ [78]. As $\kappa(M)$ increases, the accuracy of the optimal control solution trajectory may decrease. As a general rule of thumb, if the condition number $\kappa(M)$ is on the order of 10^k then it is possible to lose up to k digits of accuracy in the solution to a given linear system [78].

One method of reducing the error due to numerical conditioning of the continuation method linear system is to construct preconditioner matrices. Given a linear system of the form $M\mathbf{y} = \mathbf{b}$, a scaled system can be constructed by pre- and post-multiplying the system matrix by two square matrices, D_1 and D_2. By denoting the scaled matrix with $\hat{M}$ a new linear system can be solved,

$$\hat{M}\hat{\mathbf{y}} = \hat{\mathbf{b}} \tag{2.47}$$

where $\hat{M} = D_1 M D_2$, $\hat{\mathbf{y}} = D_2^{-1}\mathbf{y}$, and $\hat{\mathbf{b}} = D_1\mathbf{b}$. By chosing D_1 and D_2 to reduce

the condition number of the linear system given in Eq. (2.47), the solution to the original system can be computed on the better conditioned system.

Because computational speed is a major concern in the sampling methods discussed in this chapter, the method for choosing the preconditioning matrices must be computationally efficient. Additionally, as these methods should be used on general nonlinear dynamic systems, there is no assumption made on the structure of the system matrix M such as symmetry and positive-definiteness. Ruiz developed a useful preconditioning scheme for computing diagonal matrices D_1 and D_2 [79]. This iterative matrix equilibriation algorithm for computing the preconditioner matrices attempts to equilibriate the infinity norms of both the row and columns of the matrix to one. Furthermore, the algorithm shows fast linear convergence with an asymptotic rate of 1/2 and preserves the numerical structure of the original system matrix [79]. As the norm of the rows and columns approach one, the condition number also approaches one.

There are additional benefits of this preconditioner algorithm used in conjunction with the continuation framework. From one time horizon T to the next, the M matrix is marginally changing. Because the Ruiz algorithm is iterative, the initial values of D_1 and D_2 at the current time horizon T may be set as the converged values of D_1 and D_2 at the previous time horizon. This allows for fast convergence of the Ruiz algorithm as the initial values are chosen to warm-start the iteration.

An additional benefit of this preconditioner scheme is its ability to better condition systems with ill-conditioned dynamic system states. For example, describing the inertial position and velocity of an object near geosynchronous orbit around Earth requires the position scale ($r \approx 42164$ km) to be approximately 4 orders of magnitude larger than the velocity scale ($v \approx 3$ km/s) when SI units are used. To better condition the dynamics, it is possible to scale time and scale states such that the variations of the state coordinates are on the same order of magnitude. This is equivalent to a change of units to a better numerically conditioned set.

For example, describing the inertial position and velocity of an object near geosynchronous orbit around Earth requires the position scale ($r \approx 6.6$ DU) to be approximately 1 order of magnitude larger than the velocity scale ($v \approx 0.4$ DU/TU) when Earth canonical units are used [80]. By equilibriating both the rows and columns of the system matrix M, the Ruiz algorithm helps alleviate problems due to ill-conditioned system dynamics.

2.8 Results

The following system demonstrations are provided to illustrate the computational approach developed in this chapter.

2.8.1 Single DOF Double Integrator

A single degree of freedom (DOF) double integrator system is one of the most well-known optimal control problems for minimum-time problems. The forward reachable set given circular initial condition boundary conditions has a known, analytic solution. The dynamics and initial condition set for this problem are given as

$$\begin{aligned} \ddot{x} &= u, |u| \leq 1 \\ V(\mathbf{x}_0, t_0) &= \begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix}^T \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix} - 1 \leq 0 \end{aligned} \tag{2.48}$$

Figure 2.6 displays the forward reachable set for this system for a time horizon of 2 time units. These regions represent the set of possible states at the final time as the time horizon is increased from 0 to 2 time units. It should be noted that the forward reachable set in this case contains point solutions with negative optimal control Hamiltonian values at early time horizons. As a result, the reachable set shrinks in size in the top-left and bottom-right regions of the reachable set at the earlier time horizons.

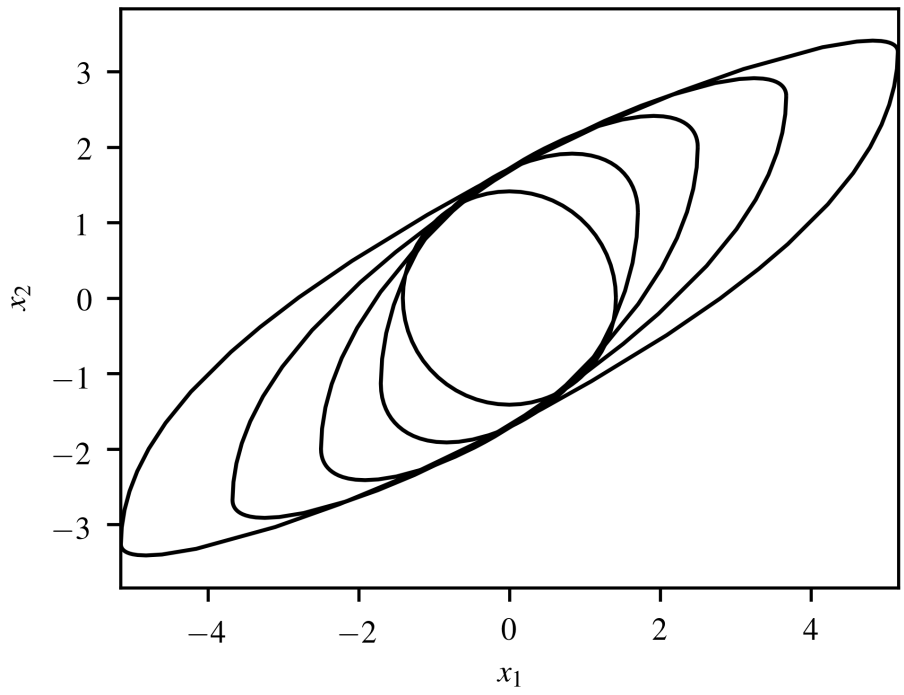


Figure 2.6: Single DOF double integrator forward reachable set at times T = 0, 0.5, 1, 1.5, and 2 (after refinement steps)

To compute this reachable set, $\theta \in [0, 2\pi)$ is evenly sampled 60 times. For each θ, the corresponding search direction vector $\hat{\mathbf{d}}_s$ was computed and the continuation method outlined in Eq. (2.32) without optimal control Hamiltonian constraints was performed to generate point solutions on the forward reachable set. For this problem, the forward reachable set was computed in about 7 seconds on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor with 8 GB 2133 MHz RAM.

2.8.2 Zermelo's Problem

Zermelo's problem is a classic optimal control problem that involves a ship on water traveling through a region of wind or water currents. The wind or water current flow field as a function of ship position is known and the ship has a known maximum speed. The problem

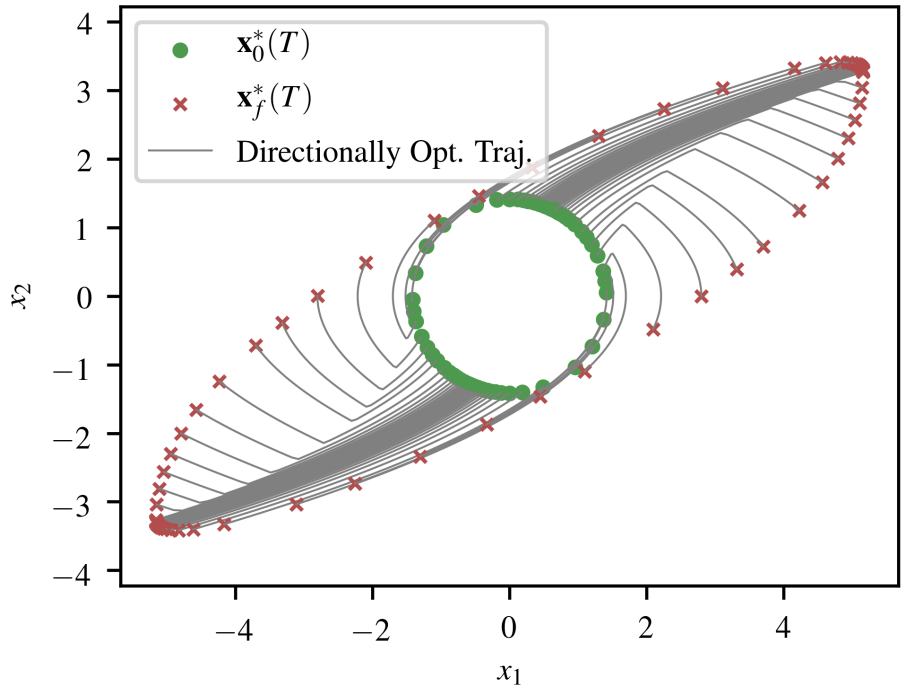


Figure 2.7: Single DOF double integrator point solution trajectories at $T = 2$ for forward reachable set (after refinement steps)

is to control the steering angle of the ship to navigate. The Zermelo dynamics are given by

$$\begin{aligned} \dot{x} &= V_m \cos(\theta) + w_x(x, y) \\ \dot{y} &= V_m \sin(\theta) + w_y(x, y) \\ \theta &\in [0, 2\pi) \end{aligned} \tag{2.49}$$

where V_m denotes the maximum velocity of the ship, θ represents the control variable of steering angle relative to the fixed x-coordinate axis, and w_x, w_y denote the velocity components of the wind/current flowfields.

The dynamics given above are nonlinear and are not affine in control. Even in this case, as long as the flowfield is C^2, there is an analytic, closed-form solution to the optimal steering law for the minimum-time problem from specified starting and ending locations [72]. This optimal steering law, called Zermelo's equation, is an ordinary differential equation in terms of the optimal steering angle and the directional derivatives of the flowfield.

To demonstrate the outlined reachability algorithm on this problem, the following ellipsoidal initial condition constraint set is used and the following dynamics parameters are chosen.

$$\begin{aligned} g_1(\mathbf{x}_0) = 9(x_0+1)^2 + 100y_0^2 \leq 1 \\ V_m = 1 \\ w_x = 1 \\ w_y = x^2 \end{aligned} \tag{2.50}$$

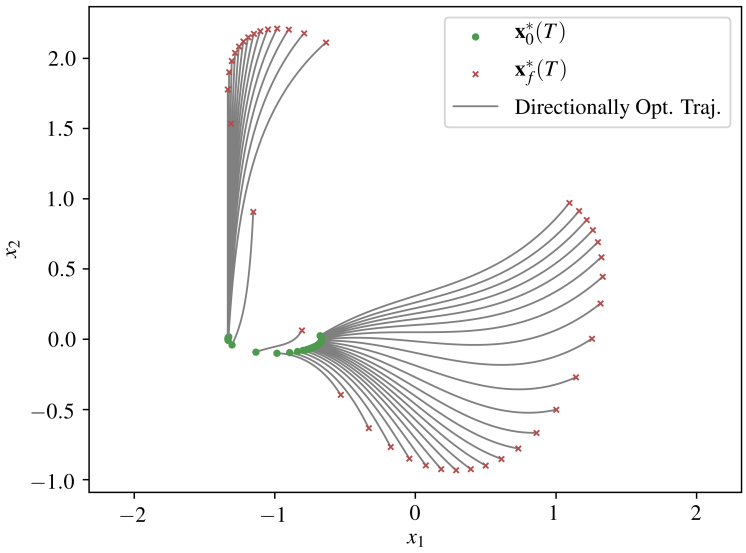


Figure 2.8: Zermelo Problem Forward Reachable Set (FRS) Samples with corresponding optimal trajectories ($T = 1$) for initial condition set given by $g_1(\mathbf{x}_0)$. The red crosses denote the samples of the reachable set, green circles denote the corresponding initial condition set samples, and the gray lines show the corresponding optimal trajectories.

Given a time horizon of 1 unit, the forward reachable set is computed using the techniques outlined above. Figure 2.8 displays the forward reachable set samples for this system as well as the corresponding extremal trajectories. To compute this reachable set, $\hat{\mathbf{d}}_s$ is uniformly sampled on the unit sphere 40 times. For each $\hat{\mathbf{d}}_s$, the continuation method outlined in Eq. (2.35) without optimal control Hamiltonian constraints was performed to generate point solutions on the forward reachable set. For this problem, the forward reachable set was computed in 1.3 seconds on a single-core of a MacBook Pro 2.3 GHz Intel Core i5

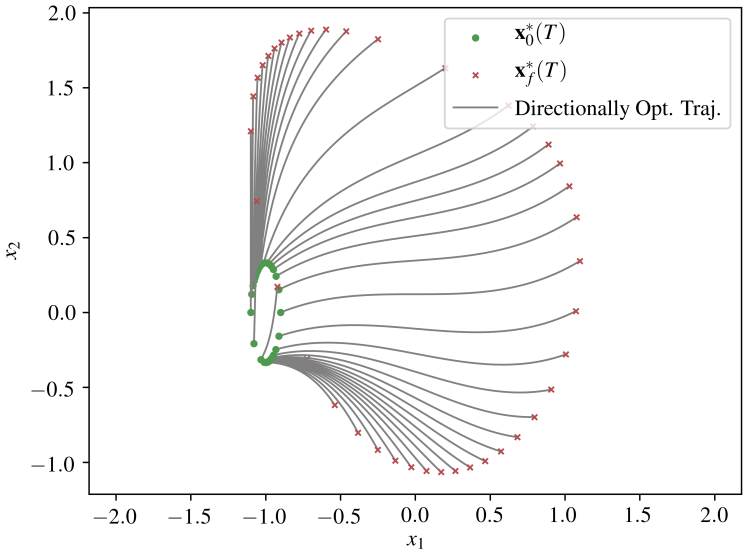


Figure 2.9: Zermelo Problem Forward Reachable Set (FRS) Samples with corresponding optimal trajectories ($T = 1$) for initial condition set given by $g_2(\mathbf{x}_0)$. The red crosses denote the samples of the reachable set, green circles denote the corresponding initial condition set samples, and the gray lines show the corresponding optimal trajectories.

processor with 8 GB 2133 MHz RAM.

If instead the initial condition constraint were given by

$$g_2(\mathbf{x}_0) = 100(x_0 + 1)^2 + 9y_0^2 \leq 1 \tag{2.51}$$

the resulting forward reachable set is shown in Figure 2.9. This reachable set was computed in 0.5 seconds for the same time horizon.

To compute the forward reachable set of the initial condition set created from $g_1(\mathbf{x}_0) \cup g_2(\mathbf{x}_0)$, the union of the corresponding forward reachable sets is computed. The results of this are shown in Figure 2.10 and Figure 2.11.

2.8.3 Orbital Relative Motion

This approach of computing reachable volumes through sampling methods is demonstrated on the case of objects in Keplerian orbit about the Earth. The exact nonlinear relative

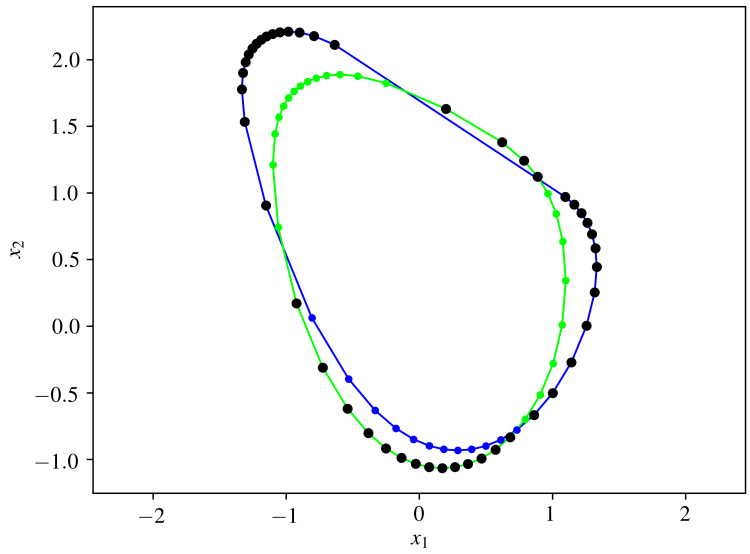


Figure 2.10: Zermelo Problem Forward Reachable Set (FRS) Samples ($T = 1$) for initial condition set given by union of $g_1(\mathbf{x}_0)$ and $g_2(\mathbf{x}_0)$. The original reachable set for $g_1(\mathbf{x}_0)$ is in blue, the original reachable set for $g_2(\mathbf{x}_0)$ is in green, and the samples that comprise the union are in black.

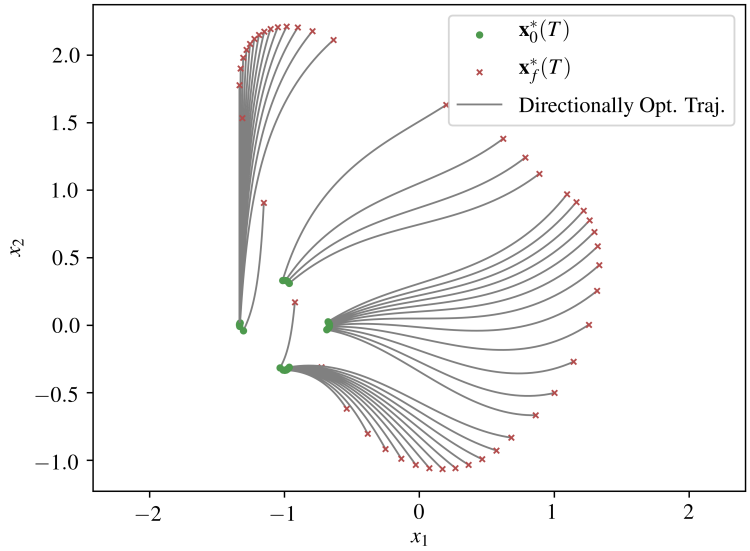


Figure 2.11: Zermelo Problem Forward Reachable Set (FRS) Samples with corresponding optimal trajectories ($T = 1$) for initial condition set given by union of $g_1(\mathbf{x}_0)$ and $g_2(\mathbf{x}_0)$. The red crosses denote the samples of the reachable set, green circles denote the corresponding initial condition set samples, and the gray lines show the corresponding optimal trajectories.

equations of motion for an object about a given arbitrary reference orbit $\mathbf{x}_r(t)$ are

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ 2\dot{f}_r\left(\dot{y} - y\frac{\dot{r}_r}{r_r}\right) + x\dot{f}_r^2 + \frac{\mu}{r_r^2} - \frac{\mu}{r^3}(r_r + x) + u_x \\ -2\dot{f}_r\left(\dot{x} - x\frac{\dot{r}_r}{r_r}\right) + y\dot{f}_r^2 - \frac{\mu}{r^3}y + u_y \\ -\frac{\mu}{r^3}z + u_z \end{bmatrix} \tag{2.52}$$

where the true anomaly rate $(\dot{f}_r)$, reference radius (r_r), and reference radius time derivative $(\dot{r}_r)$ can be directly computed using Keplerian dynamics and the inertial radius of the spacecraft is r, defined as $r = \sqrt{(r_r + x)^2 + y^2 + z^2}$ [80]. These equations of motion represent the relative motion between an object and another reference object in a reference orbit. The dynamics are expressed in a rotating Hill frame, where the radial axis (x) points from the center of the Earth to the reference object and the along-track axis (y) is defined as perpendicular to the radial vector and is positive in the direction of the reference orbit velocity.

The discussed approach is demonstrated on the case of a spacecraft in an eccentric geostationary transfer orbit (GTO) with maximum thrust constraints. In this scenario, one wants to know the set of feasible starting positions around a nominal GTO apoapsis point such that a controlled spacecraft with maximum thrust constraints can converge to within 10 meters in position and 0.1 meters per second in velocity by the time it reaches the nominal GTO periapsis point. This is useful information in designing a thruster system in terms of thruster layout and capability when the nominal orbit is a GTO. This computation is achieved by computing the position subspace backwards reachable set with the state constraint set defined at the GTO apoapsis point.

For this demonstration, the terminal condition set is given by

$$g(\mathbf{x}_0) = \mathbf{x}_0^T E \mathbf{x}_0 \leq 1 \tag{2.53}$$

where $E = \text{diag}(10, 10, 10, 0.1, 0.1, 0.1)$ and the feasible control set as

$$\|\tilde{\mathbf{u}}\|_5 \leq 1 \tag{2.54}$$

where $u_i = 1\text{e-}6\ \tilde{u}_i, i = 1, 2, 3$. The nominal GTO orbit has a periapsis radius of 7000 km and an apoapsis radius at GEO with 42164 km.

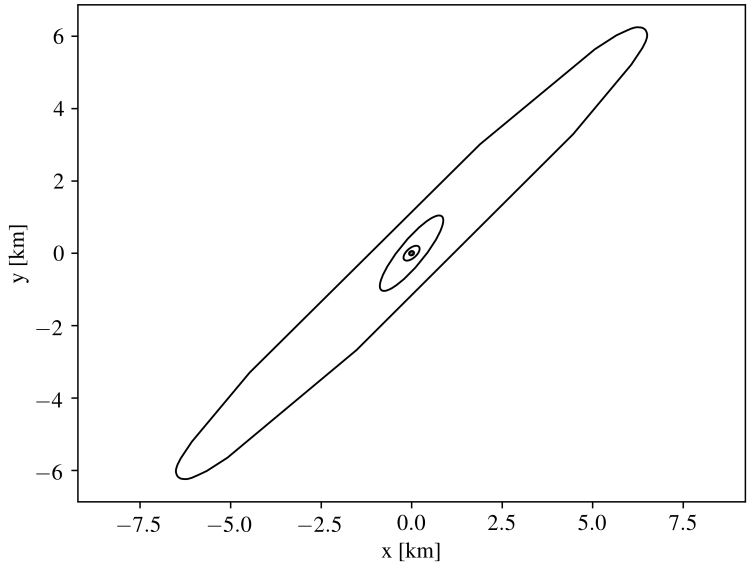


Figure 2.12: x, y position subspace backwards reachable set for 3-DOF nonlinear relative Keplerian motion set at true anomaly, $\nu = [\frac{1}{4}\pi, \frac{1}{2}\pi, \frac{3}{4}\pi, \pi]$ in rotating Hill frame - GTO orbit

The results of this analysis are shown in Figure 2.12 which took 230 seconds to compute on the previously mentioned Mac machine. Looking at these results, one can see that it is much easier to get to the nominal periapsis state when the starting positions have a positive correlation between the radial and along-track components. Otherwise, the orbital motion disallows much of the position states with negative correlations with radial and along-track components.

Furthermore, the computed backwards reachable set does not decrease in size as the time horizon is increased. As a result, the backwards reachable tube, $\mathcal{R}_t^-$, is identical to the backwards reachable set, $\mathcal{R}^-$, for computed time horizons. This also suggests that once a position state is backwards reachable, it remains backwards reachable for the remainder of the selected time horizon.

Low-Thrust Position Reachability in Low Earth Orbit

The discussed approach is first demonstrated on the case of a spacecraft in circular Low Earth orbit (LEO) with maximum thrust constraints where only planar motion is considered. The two-dimensional forward position subspace reachable set is then computed using the outlined methodology. This kind of analysis is useful in determining the maneuverability of a low-thrust spacecraft in LEO over a single orbit. This problem is also equivalent to finding reachable positions for an uncontrolled spacecraft subject to unmodeled, external disturbances of a particular magnitude.

As the reachable set for the two-dimensional subspace is explored with a one-dimensional search (θ), this example demonstrates subspace reachability calculations with computations $\mathcal{O}(k)$ as opposed to $\mathcal{O}(k^4)$ or $\mathcal{O}(k^3)$. Furthermore, this example demonstrates the capability to compute the reachability set boundaries under nonlinear dynamics using sampling methods.

The initial reachability set represented by $V(\mathbf{x}_0, t_0)$ is chosen to be defined by the ellipsoidal constraint

$$V(\mathbf{x}_0, t_0) = \begin{bmatrix} \mathbf{d}_0 \\ \mathbf{v}_0 \end{bmatrix}^T \begin{bmatrix} \frac{1}{r_d^2}\mathbb{I} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \frac{1}{r_v^2}\mathbb{I} \end{bmatrix} \begin{bmatrix} \mathbf{d}_0 \\ \mathbf{v}_0 \end{bmatrix} - 1 \leq 0 \tag{2.55}$$

with r_d set to 1 meter and r_v set to 0.1 meters per second. The control has an upper bound of $u_m =$ 2e-5 m/s^2, which is equivalent to a 500 kg spacecraft with a 0.01 N thruster. The

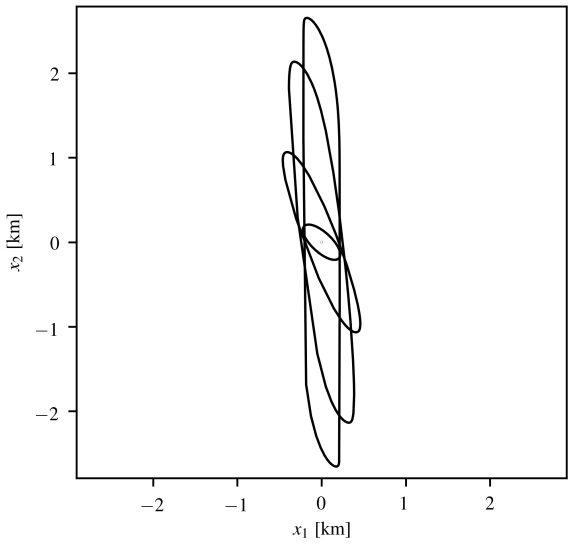


Figure 2.13: Position subspace forward reachable set for 2-DOF nonlinear relative Keplerian motion set at times $T = [\frac{1}{4}P, \frac{1}{2}P, \frac{3}{4}P, P]$ in rotating Hill frame - LEO orbit

orbital altitude is initially set at 400 km.

Figure 2.13 shows the set of reachable positions for a low-thrust spacecraft in LEO at given fractions of an orbit. The initial number of θ point solutions is 60 and 60 additional points solutions were spawned in the mesh refinement process taking 293 seconds to compute.

In this orbital regime, the control authority is relatively weak compared to the local dynamics. As a result, the reachable set is primarily driven by the periodic dynamics as opposed to the optimal control. Because this system is not STLC, there are certain states that are reachable at one time and not reachable at subsequent times.

Unsafe Positions in Geosynchronous Orbit

This demonstration involves the computation of a position backwards reachable set for an object in circular geosynchronous orbit (GEO) for a single orbit period. If the terminal boundary condition set is considered unsafe, this backwards reachable set represents unsafe initial conditions. This analysis then determines the set of unsafe initial positions that can lead to the unsafe terminal set, making it useful for safety assurance in collision avoid-

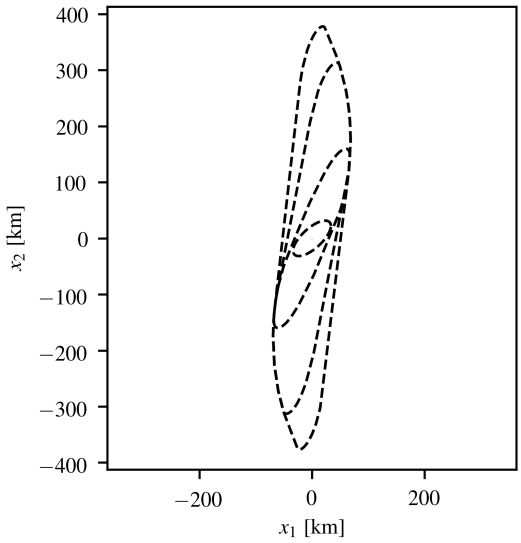


Figure 2.14: Position subspace backwards reachable tube for 2-DOF nonlinear relative Keplerian motion set at times $T = [\frac{1}{4}P, \frac{1}{2}P, \frac{3}{4}P, P]$ in rotating Hill frame - GEO orbit

ance scenarios. This problem is also equivalent to finding unsafe initial positions for an uncontrolled spacecraft subject to worst-case unmodeled, external disturbances of a particular magnitude. Additionally, because states that are unsafe at one time horizon cannot be made safe at a larger time horizon, the backwards reachable tube is computed.

The initial reachability set represented by $V(\mathbf{x}_0, t_0)$ is chosen to be defined by the ellipsoidal constraint given in Eq. (2.55) with r_d set to 50 meters and r_v set to 1 meters per second. This corresponds to approximately a 50 meter keep-out circle for the positions with an entrance velocity of about 1 meter per second.

In this case, the maximum control input magnitude was selected to be the same order of magnitude as the non-Keplerian orbital perturbations such as J_2 effects and lunar gravitational forces [80], u_m = 1e-5 m/s^2. This represents the situation of a worst-case unmodeled disturbance that attempts to drive the system state towards the unsafe region in state space.

Figure 2.14 shows the set of unsafe spacecraft initial positions that can be driven to the unsafe set (representing a collision) by worst-case disturbances. The initial number of θ point solutions is 60 and 48 additional points solutions were spawned in the mesh refinement

process taking 287 seconds to compute the backwards reachable tube. It should be noted the backwards reachable tube in Figure 2.14 does not decrease in size as the time horizon is increased. As a result, the backwards reachable set at time T_1, $\mathcal{R}_t^-(T_1)$, is completely contained within the backwards reachable set at time T_2, $\mathcal{R}_t^-(T_2)$ given $T_2 \geq T_1$.

2.8.4 Six DOF Quadrotor Model

Finally the proposed technique is demonstrated on a 6-DOF nonlinear quadrotor model derived by Beard [81], eq. (16)-(19)). This 12-dimensional model is commonly used as a nonlinear dynamic system verification benchmark problem [82]. The benchmark problem in literature is not recreated in this thesis because the benchmark exactly specifies the initial Euler angles and Euler angle rates. Future work will extend the capabilities of the outlined methodology defining optimality conditions for various initial condition sets of this type. Furthermore, the benchmark problem definition includes closed loop PD feedback control for the height, roll, and pitch variables in a system verification framework while this demonstration examines system capability driven by optimal control. For the sake of comparison with the benchmark problem results from other reachability algorithms, the chosen parameters are the same as the benchmark and the current algorithm is implemented on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor with 8 GB 2133 MHz RAM.

Using the notation given by Beard, the following parameter values are chosen: g = 9.81 [m/s^2], R = 0.1 [m], l = 0.5 [m], M_{rotor} = 0.1 [kg], M = 1 [kg], and $m = M + 4M_{rotor}$. For the 12-dimensional state vector, the first three states correspond to the inertial center of mass position variables in the north, east, and up directions. The remaining states include the body frame velocity, Euler angles from the vehicle to body frame, and body frame Euler angle rates. The control inputs are torque inputs along the body axes of the quadrotor.

In this example, the following initial condition set and feasible control set is used.

$$g(\mathbf{x}_0) = \mathbf{x}_0^T E \mathbf{x}_0 \leq 1 \tag{2.56}$$

where $E = \text{diag}(0.4\ \mathbb{I}_6, 0.1\ \mathbb{I}_6)$ and the feasible control set as

$$\|\tilde{\mathbf{u}}\|_{10} \leq 1 \tag{2.57}$$

where $\tilde{u}_i = 2\text{e}5\ u_i\ , i = 1, 2, 3.$

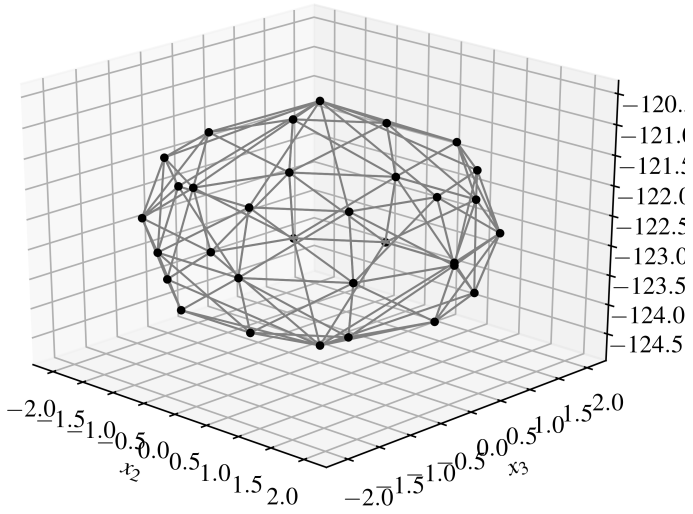


Figure 2.15: x, y, z position subspace backwards reachable set for 6-DOF quadcopter model after 5 seconds

The results of this analysis are shown in Figure 2.15 which took 41.3 seconds to compute on the previously mentioned Mac machine with 32 samples. As the forward reachable set for the 3-dimensional subspace is explored with a 2-dimensional search of $\boldsymbol{\theta}$, this example demonstrates subspace reachability calculations with computations $\mathcal{O}(k^2)$ as opposed to $\mathcal{O}(k^{12})$ or $\mathcal{O}(k^{11})$.

2.9 Conclusions

This chapter demonstrates the capability to compute convex reachable sets for both forward and backward time using sampling methods. Using continuation methods on the first-order necessary conditions of optimality, the subspace reachability problem may be recast into an initial value problem. Within numerical precision, these samples lie on the boundary of the convex hull of the subspace reachable sets. To suppress the numerical integration error in the continuation methods, the matrix equilibriation technique is described. Furthermore, the introduction of Newton-type corrective steps ensure the overall necessary condition of optimality constraint is sufficiently satisfied. With the inclusion of optimal control Hamiltonian constraints, reachable tubes are computed.

While previous continuation-based reachability techniques used ellipsoidal initial condition and feasible control sets, this concept was extended to capture affine transformations of normed vector space unit balls. This chapter also showed that reachable volumes generated from unions of initial condition sets can be computed by performing unions on the individual reachable volumes.

CHAPTER 3

DECENTRALIZED TECHNIQUES FOR SAMPLING OF SUBSPACE REACHABLE SETS

3.1 Curvature-Based Sampling

In differential geometry, the Gauss map continuously maps a hypersurface $M \in \mathbb{R}^n$ to the unit sphere $S \in \mathbb{S}^{n-1}$ [83]. Said differently, the Gauss map maps a point on a surface to the unit vector normal to the surface at that point.

With the current formulation, the inverse relationship is used ($S \rightarrow M$) because the reachable set boundary is parametrized by the unit normals, $\hat{\mathbf{d}}_s(\boldsymbol{\theta})$. As a result of using unit normals to describe a hypersurface, a convex hull of the true hypersurface is guaranteed [16, 83]. This also supports the discussion earlier that the reachable sets resulting from this formulation are convex representations of full state reachable sets projected on the subspace of interest [10].

Definition 1: Support Function

The support function is used commonly in convex geometry applications where it describes a surface by the signed distance of its tangent planes to the origin. The support function can be expressed as a function of the surface normal $\hat{\mathbf{d}}$ as

$$h(\hat{\mathbf{d}}) = \sup\{\hat{\mathbf{d}} \cdot \mathbf{x}(\hat{\mathbf{d}}) : \mathbf{x} \in M\}, \hat{\mathbf{d}} \in \mathbb{S} \tag{3.1}$$

where M is a hypersurface/manifold embedded in $\mathbb{R}^n$ and $\mathbb{S}^{n-1}$ denotes the unit hypersphere.

Given the support function, a point on the surface may be decomposed as

$$\mathbf{x}(\hat{\mathbf{d}}) = h(\hat{\mathbf{d}})\hat{\mathbf{d}} + \nabla_{\mathbb{S}} h(\hat{\mathbf{d}}) \tag{3.2}$$

where $\nabla_{\mathbb{S}}$ indicates the intrinsic gradient with respect to the unit sphere [84]. Finally, the envelope operator may be defined as

$$\mathcal{E} : C^1(\mathbb{S}, \mathbb{R}) \rightarrow C(\mathbb{S}, \mathbb{R}^n) : h \rightarrow \mathbf{x} \tag{3.3}$$

which defines a mapping between the support functions and the corresponding surface.

A benefit of this surface representation is that geometric properties can be calculated when the support function is known. For example, the Weingarten map or shape operator of this surface is given by

$$W = -[H_{\mathbb{S}}(h) + h\mathbb{I}]^{-1} \tag{3.4}$$

where $H_{\mathbb{S}}(h)$ is the hessian of h with respect to the unit sphere. Once this is computed, the principal curvatures are related to the eigenvalues of $H_{\mathbb{S}}(h)$ through $\kappa_i = -(\lambda_i + h)^{-1}$. Common curvature metrics include the mean curvature $\overline{\kappa} = \frac{1}{2}Tr[W]$ and the Gaussian curvature $K = \det(W)$ [83, 84].

In Sampoli et al., the authors achieve curvature-dependent sampling by applying the envelope operator to a set uniformly distributed points on the unit sphere [84]. This result is expanded upon here.

Proposition 1: Curvature-dependent sampling from uniform sampling of unit sphere and support function expression of manifold

Given an envelope operator $\mathcal{E}$ associated with hypersurface support function $h(\mathbf{d})$ and unit hypersphere $\mathbb{S}^{n-1}$ embedded in $\mathbb{R}^n$, if the sampling of N unit vectors from the unit hypersphere $\mathbf{d} \in \mathbb{S}^{n-1}$ is equidistant such that $\|\mathbf{d}_i - \mathbf{d}_j\| = \gamma > 0 \ \forall i, j = 1...N, j \in \mathcal{N}(i)$ where

$\mathcal{N}(i)$ denotes the set of neighboring samples to $\mathbf{d}_i$, the resulting distribution of samples on the manifold $\mathbf{x} \in M \subset \mathbb{R}^n$ is curvature-dependent where $\|\mathbf{x}_i - \mathbf{x}_j\| \propto \|W(\mathbf{d}_i)^{-1}\|$.

Proof: From Eq. (3.2),

$$\begin{aligned}
\frac{\partial \mathbf{x}}{\partial \mathbf{d}} &= h(\mathbf{d})\mathbb{I}_n + \mathbf{d}\frac{\partial h}{\partial \mathbf{d}}(\mathbf{d}) + \frac{\partial^2 h}{\partial \mathbf{d}^2}(\mathbf{d}) \\
&\quad - \frac{\partial h}{\partial \mathbf{d}}(\mathbf{d})\mathbf{d}\mathbb{I} - \mathbf{d}\frac{\partial h}{\partial \mathbf{d}}(\mathbf{d}) - \mathbf{d}\mathbf{d}^T\frac{\partial^2 h}{\partial \mathbf{d}^2}(\mathbf{d}) \\
&= h(\mathbf{d})\mathbb{I}_n + \frac{\partial^2 h}{\partial \mathbf{d}^2}(\mathbf{d}) - \mathbf{d}\mathbf{d}^T\frac{\partial^2 h}{\partial \mathbf{d}^2}(\mathbf{d}) \\
&= h(\mathbf{d})\mathbb{I}_n + H_{\mathbb{S}}(h(\mathbf{d})) \\
&= -W(\mathbf{d})^{-1}
\end{aligned} \tag{3.5}$$

Performing a first order Taylor series approximation of $\mathbf{x}(\mathbf{d})$,

$$\begin{aligned}
\mathbf{x}(\mathbf{d} + \delta\mathbf{d}) - \mathbf{x}(\mathbf{d}) = \delta\mathbf{x} &\approx \frac{\partial \mathbf{x}}{\partial \mathbf{d}}\delta\mathbf{d} \\
&= -W(\mathbf{d})^{-1}\delta\mathbf{d}
\end{aligned} \tag{3.6}$$

Consequently, for neighboring unit vector samples, $\mathbf{d}_i$ and $\mathbf{d}_j$, the following expression holds

$$\begin{aligned}
\|\mathbf{x}_j - \mathbf{x}_i\| &\approx \| - W(\mathbf{d}_i)^{-1}\delta\mathbf{d}\| \\
&\leq \|W(\mathbf{d}_i)^{-1}\|\|\mathbf{d}_j - \mathbf{d}_i\|
\end{aligned} \tag{3.7}$$

□

The uniformly spaced samples on the unit sphere will map to dense clusters of samples in high curvature regions, and more distributed samples in low curvature regions. As curvature of a surface is related to how the unit normal vector changes locally, uniform unit sphere samples cluster near high curvature areas and distribute near low curvature areas. This concept is illustrated in Fig. 3.1 for the mapping between a unit circle and ellipse however the results generalize to higher dimensions [84].

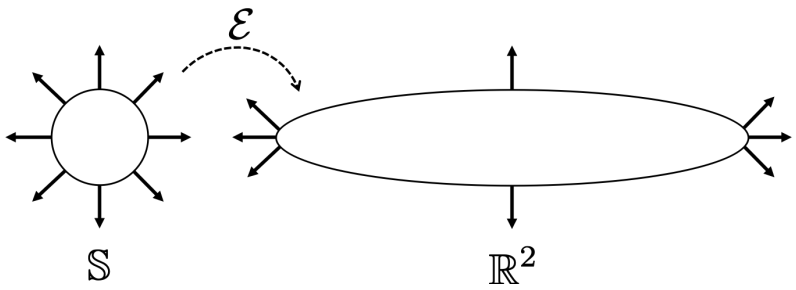


Figure 3.1: Two dimensional illustration of envelope operator and resulting curvature-based sampling

Lemma 1: Subspace Reachability Support Function

In the case of the subspace reachability problem, the support function h can be defined as

$$h(\hat{\mathbf{d}}_s) = \sqrt{2V(\hat{\mathbf{d}}_s)}, \forall\ \hat{\mathbf{d}}_s \in \mathbb{S}^{s-1} \tag{3.8}$$

Proof: From Eq. (2.2),

$$\begin{aligned} V(\mathbf{x}_f, \hat{\mathbf{d}}_s, t_f) &= \frac{1}{2}\mathbf{x}_f^T G \mathbf{x}_f \\ &= \frac{1}{2}\mathbf{x}_{sf}^T \hat{\mathbf{d}}_s \hat{\mathbf{d}}_s^T \mathbf{x}_{sf} \end{aligned} \tag{3.9}$$

Eq. (2.3) is equivalent to the following optimal control problem

$$\begin{aligned} &\sup_{\mathbf{u}\in U} \hat{\mathbf{d}}_s^T \mathbf{x}_{sf} \\ s.t.\quad &\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \\ &\mathbf{g}(\mathbf{x}_0, t_0) = \mathbf{0} \end{aligned} \tag{3.10}$$

because V and $\sqrt{V}$ are both monotonically increasing with increasing V. In this new optimal control problem, the state $\mathbf{x}_{sf}$ still maximizes extent along $\hat{\mathbf{d}}_s$ while satisfying dynamics and reachability constraints. Consequently, the support function for the subspace

reachability problem can be expressed as a function of $\hat{\mathbf{d}}_s$ as

$$h(\hat{\mathbf{d}}_s) = \sqrt{2V(\hat{\mathbf{d}}_s)}, \forall\ \hat{\mathbf{d}}_s \in \mathbb{S}^{s-1} \tag{3.11}$$

where $V(\hat{\mathbf{d}}_s)$ is the optimal control performance index defined in Eq. (2.2).

□

With this result, a uniform distribution of initial search directions directly results in a curvature-based sampling. For two- and three-dimensional subspace reachable set problems ($\hat{\mathbf{d}}_s \in \mathbb{S}, \mathbb{S}^2$), efficient analytical methods for computing a set of uniform samples from on unit sphere are known [85]. These methods are used in the examples presented in this chapter. Efficient methods are also available for higher dimensional problems [86, 85]. These methods provide computationally independent samples of the unit sphere which still allows for parallelizability.

3.2 Distance-Based Sampling

As the reachable set time horizon increases, dense or sparse densities of point solutions may form on the reachable set boundary. However, quick rendering of an implicit surface benefits from uniform spacing between points [35, 36]. This section discusses two classes of techniques that use distance metrics to distribute point solutions on the reachable set boundary: one that changes the total number of samples and one that does not.

3.2.1 Distributed Control

Systems in which multiple independent agents share information with other networked agents and perform cooperative tasks are called distributed networked multi-agent systems [87]. In this framework, each agent is subject to local dynamics, has a communication

topology modeled using graph theory, and must execute a control policy in pursuit of the network's goals [88]. This communication network can be represented using an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where $\mathcal{G}$ is defined with vertices $\mathcal{V}$ and edges $\mathcal{E}$ [40]. In this context, the individual point solutions that sample the reachable set are vertices of the graph. The set of edges of the graph are defined by connecting each particle with it's nearest neighbors.

By treating each point solution as an agent in a distributed system, it is possible to adjust the surface distribution by adjusting the individual locations of the point solutions to increase the surface uniformity of the entire group. Furthermore, these agents do not necessarily need information from every other agent, but rather only information from the neighboring agents. Controlling multi-agent distributed networks is widely studied, with techniques including potential fields, leader agents, and system-wide energy minimization [89, 90, 91]. Moreover, multi-agent distributed control has been applied to the problem of uniform coverage of regions [92, 93].

If the time horizon is fixed, the reachable set boundary coverage problem may be recast as a multi-agent distributed control problem.

Definition 2: Reachability Surface Mesh States

By concatenating each of the p point solutions into a single state, one can acquire a centralized, global graph state denoted by

$$\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x}_1^T & \mathbf{x}_2^T & \dots & \mathbf{x}_p^T \end{bmatrix}^T \tag{3.12}$$

where $\tilde{(\cdot)}$ denotes the concatenation of all the individual point solution quantities.

To quantify surface sampling density of a graph, a pseudo-potential energy function based relative positions between neighboring particles is employed. The following sections discuss how the potential energy functions can be utilized to achieve more uniform surface sampling.

Definition 3: Graph potential energy based on particle positions

In general, a suitable global potential energy is defined as

$$J = \sum_{i=1}^{p} \sum_{j \in \mathcal{N}_i} J_{ij}(\mathbf{x}_{f,i} - \mathbf{x}_{f,j}) \tag{3.13}$$

where $\mathcal{N}_i$ denotes the set of neighboring particles to particle i and J_{ij} denotes the pairwise particle energy between particle i and particle j.

Due to the minimum total potential energy principle, particles subjected to conservative forces will displace to a position that minimizes the global potential energy [94]. As a result, by moving all the particles along the negative potential energy gradient, a potential energy minimum is achieved. This natural phenomenon has been extensively studied as it has many applications in fields like thermodynamics, structural energy minimization, and computational chemistry and molecular dynamics [94, 95]. Moreover, there has been extensive study of discretizing surfaces using samples via energy minimization [34, 35, 96]. This work uses a elastic potential energy function and is described below.

However, for the problem of acquiring uniform particle densities on a surface, there are multiple potential energy functions that can be used. Potential functions that are continuous and scale invariant with respect to inter-particle distance are better for this problem [34]. In this chapter, the Laplacian cost function J defined in Eq. (3.14) is used [40] as a pseudo-potential energy function, analogous to a Lyapunov function interpretations from dynamical system stability [97]. In this formulation, the Laplacian cost is only defined in the subspace of interest with respect to the reachability problem. This is because the distribution of state components outside of the subspace of interest should not affect the uniformity metric for samples of the subspace reachable set.

Definition 4: Subspace Laplacian cost function

The distance-based Laplacian cost is defined as

$$\begin{aligned} J &= \sum_{ij\in\mathcal{E}} (\mathbf{x}_{f,i} - \mathbf{x}_{f,j})^T Q^T Q (\mathbf{x}_{f,i} - \mathbf{x}_{f,j}) \\ &= \sum_{ij\in\mathcal{E}} (\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j})^T Q_s^T Q_s (\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j}) \\ &= \sum_{ij\in\mathcal{E}} d(\mathbf{x}_{f,i}, \mathbf{x}_{f,j})^2 \\ Q &= \begin{bmatrix} Q_s & \mathbf{0}_{s\times(n-s)} \end{bmatrix} \end{aligned} \tag{3.14}$$

where $Q_s \in \mathbb{R}^+_{s\times s}$, is a user-specified symmetric positive definite matrix that appropriately scales the coordinates of the state for the s components of the final subspace state $\mathbf{x}_{sf}$. This allows the Laplacian cost function to be succintly expressed as

$$J = \tilde{\mathbf{x}}_f^T L \tilde{\mathbf{x}}_f = \tilde{\mathbf{x}}_{sf}^T L_s \tilde{\mathbf{x}}_{sf} \tag{3.15}$$

where $L \in \mathbb{R}_{np\times np}$ is the weighted Laplacian matrix [40].

$$L_{ij} = \begin{cases} -Q^T Q & \text{if } i \in \mathcal{N}_j \\ Q^T Q \deg(i) & \text{if } i = j \\ \mathbf{0}_n & \text{otherwise} \end{cases} \tag{3.16}$$

where $\deg(i)$ the number of neighbors for the i_{th} point solution and $\mathcal{N}_j$ denotes the set of neighbors of the j_{th} point solution. The subspace counterpart of the Laplacian, $L_s \in \mathbb{R}_{sp\times sp}$, is given by

$$L_{s,ij} = \begin{cases} -Q_s^T Q_s & \text{if } i \in \mathcal{N}_j \\ Q_s^T Q_s \deg(i) & \text{if } i = j \\ \mathbf{0}_n & \text{otherwise} \end{cases} \tag{3.17}$$

The Laplacian cost function in Eq. (3.14) is equivalent to the elastic potential energy ex-

pression of a multi-particle spring-mass system where the connections are described the graph structure. This approach of modeling interparticle interactions as elastic springs and searching for minimum energy configurations is commonly utilized in molecular mechanics [95]. Furthermore, the 'spring stiffnesses' in different coordinate directions would be described using the Q_s weighting matrix.

To study this system driven by the Laplacian cost function potential field, a study of the system equilibirum (if it exists) is useful. First, we review concepts from analytical dynamics to define the necessary and sufficient conditions for system equilibrium.

In analytical dynamics, a holonomic constraint is a relation between position variables $\mathbf{r}$ which can be expressed as $\kappa(\mathbf{r}, t) = 0$. Systems that include only holonomic constraints are holonomic. If, in addition, the force field driving the system is derived from a scalar potential function, the system is conservative. From analytical dynamics, generalized coordinates are parameters that describe the configuration of the system. For holonomic systems, there is an independent generalized coordinate for every degree of freedom of the system with constraints on the motion included.

Proposition 2: Necessary and sufficient conditions for minimum potential energy of conservative holonomic system[94]

Given a dynamic system of p interacting particles with states $\mathbf{r} \in \mathbb{R}^{np}$ driven by a scalar conservative force field $U(\mathbf{r})$ such that

$$\dot{\mathbf{r}} = -\frac{dU}{d\mathbf{r}}^T$$

with holonomic constraint $\kappa(\mathbf{r}, t) = 0$, and with independent generalized coordinates $\mathbf{q}$, equivalent necessary conditions for the equilibrium of this system are given by

$$-\frac{dU}{d\mathbf{r}_i}^T \cdot \delta\mathbf{r}_i = 0, \quad i = 1, 2, ..., p \tag{3.18}$$

$$\frac{\partial U}{\partial \mathbf{q}_i}^T = \mathbf{0}, \quad i = 1, 2, ..., p \tag{3.19}$$

where $\delta \mathbf{r}_i$ denotes an arbitrary virtual displacement of the i_{th} particle and $\frac{dU}{d\mathbf{r}_i}^T$ denotes the total conservative internal force on the i_{th} particle from its neighboring particles. Furthermore, a sufficient condition for the system equilibrium is

$$\frac{\partial^2 U}{\partial \mathbf{q}_i^2}(\mathbf{q}_i^*) > 0 \text{ where } \frac{\partial U}{\partial \mathbf{q}_i}^T (\mathbf{q}_i^*) = \mathbf{0}, \quad i = 1, 2, ..., p \tag{3.20}$$

Proof: The principle of virtual work states that if a system is in static equilibirum, the virtual work (δW) of the applied forces is zero for all possible virtual displacements ($\delta \mathbf{r}$) . Virtual displacements are feasible changes in position at a fixed time which makes them a tangent vector to the constraint manifold. For the system of p particles without applied external forces,

$$\delta W = \sum_{i=1}^{p} \left[\sum_{j=1, j \neq i}^{p} \mathbf{F}_{ij} \right] \cdot \delta \mathbf{r}_i \tag{3.21}$$

where $\mathbf{F}_{ij}$ denotes the internal force between the i_{th} and j_{th} particles and $\delta \mathbf{r}_i$ indicates an arbitrary virtual displacement of the i_{th} particle. For the case of a force derived from a potential field U and $s-1$ independent generalized coordinates q_j,

$$\begin{aligned} \delta W_i &= -\frac{\partial U}{\partial \mathbf{r}_i}^T \cdot \left(\frac{\partial \mathbf{r}_i}{\partial q_{i,1}} \delta q_{i,1} + \cdots + \frac{\partial \mathbf{r}_i}{\partial q_{i,s-1}} \delta q_{i,s-1} \right) \\ &= -\frac{\partial U}{\partial q_{i,1}} \delta q_{i,1} + \cdots + -\frac{\partial U}{\partial q_{i,s-1}} \delta q_{i,s-1} \end{aligned} \tag{3.22}$$

As each term should independently vanish with arbitrary virtual displacements in Eq. (3.21) and Eq. (3.22) using the principle of virtual work, the following equivalent static equilibirum necessary conditions are found

$$-\frac{\partial U}{\partial \mathbf{r}_i}^T \cdot \delta \mathbf{r}_i = 0, \quad i = 1, 2, ..., p \tag{3.23}$$

$$\frac{\partial U}{\partial \mathbf{q}_i}^T = \mathbf{0}, \quad i = 1, 2, ..., p \tag{3.24}$$

Eq. (3.23) shows that particle i is in static equilibrium when the sum of the internal forces acting on particle i are perpendicular to the possible virtual displacements. As the set of all virtual displacements is the tangent space of the holonomic constraint manifold, the sum of internal forces on particle i must be normal to the holonomic constraint manifold at the location of particle i. Eq. (3.24) equivalently shows that static equilibrium is achieved when the potential energy is stationary with respect to the generalized coordinates, $\mathbf{q}_i$. If, in addition, the Hessian of the potential energy field is positive definite at $\mathbf{q}_i^*$

$$\frac{\partial^2 U}{\partial \mathbf{q}_i^2}(\mathbf{q}_i^*) > 0 \text{ where } \frac{\partial U}{\partial \mathbf{q}_i}^T (\mathbf{q}_i^*) = \mathbf{0}, \quad i = 1, 2, ..., p$$

then the potential energy is minimized at the system configuration specified by $\mathbf{q}_i^*, i = 1, 2, ..., p$.

□

Using this classical mechanics results, we are now going to apply this to the problem of finding minimum energy surface sampling.

Lemma 2: Uniform subspace point solution sampling in manifold tangent space

Given a dynamic system of p interacting particles with states $\tilde{\mathbf{x}}_{sf} \in \mathbb{R}^{sp}$ driven by the Laplacian cost defined in Eq. (3.14) as a scalar potential function $J(\tilde{\mathbf{x}}_{sf})$ such that

$$\frac{d\tilde{\mathbf{x}}_{sf}}{dt} = -\frac{dJ}{d\tilde{\mathbf{x}}_f}^T$$

with the constraint that $\mathbf{x}_{sf,i}$ should lie on the reachable set boundary, then $\tilde{\boldsymbol{\theta}}$ may be used as independent generalized coordinates and the dynamic system has a equilibrium condition

of

$$\deg(i)[Q_s^T Q_s \mathbf{x}_{sf,i}]_{\|} = \sum_{j \in \mathcal{N}_i} [Q_s^T Q_s \mathbf{x}_{sf,j}]_{\|}$$
$$[Q_s^T Q_s \mathbf{x}_{sf,i}]_{\|}^* = \frac{1}{\deg(i)} \sum_{j \in \mathcal{N}_i} [Q_s^T Q_s \mathbf{x}_{sf,j}]_{\|}$$

where $(\cdot)_{\|}$ denotes the projection of a vector onto the tangent space to the surface at that state, $T_{\mathbf{x}_{sf,i}} M$. This result shows that equilibrium condition for the subspace states is for each weighted point solution to be the average of all of its weighted neighbors within $T_{\mathbf{x}_{sf,i}} M$.

Proof: The pairwise gradient of the Laplacian cost with respect to the subspace states is given as

$$\frac{\partial J_i}{\partial \mathbf{x}_{sf,j}}^T = Q_s^T Q_s (\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j})$$
$$= Q_s^T Q_s \|\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j}\| \frac{(\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j})}{\|\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j}\|}$$

which shows that the derived force from the Laplacian cost is central and radially symmetric. As the Laplacian cost is defined in terms of relative position only, the force derived from this scalar potential field is conservative as the work done by the force is path independent [94]. Furthermore, the Hessian of this Laplacian cost with respect to the final subspace state is

$$\frac{\partial^2 J_i}{\partial \mathbf{x}_{sf,i}^2} = Q_s^T Q_s > 0$$

Because the angle parameter and final subspace state share a one-to-one mapping, the Hessian of the Laplacian cost with respect to the angle parameters is also positive definite.

The reachable set boundary at the current time horizon can be expressed as $\bar{V}(\mathbf{x}_f, T) = 0$

where $\bar{V}$ denotes the value function from optimal control theory. This is because the zero level sets of the value function over time represent the boundary of the minimum time reachable set [7, 8]. Expressed in this manner, one can view the reachability level set as a holonomic constraint for the final states. As a result, it is useful to treat this system of final subspace states as a constrained, conservative holonomic system.

The holonomic constraint for the subspace reachable set boundary reduces the degrees of freedom of the system by one. In this formulation, an appropriate set of generalized coordinates is the $\boldsymbol{\theta} \in \mathbb{R}^{s-1}$ parameters. As previously shown, these angular parameters uniquely define a sample on the subspace reachable set.

As this subspace reachability system can be identified as a conservative holonomic dynamic system, it is very useful to use results from Proposition 2. For the subspace reachable set sampling problem, the necessary condition in Eq. (3.23) is equivalent to

$$\frac{\partial J}{\partial \mathbf{x}_{sf,i}}^T = \sum_{j \in \mathcal{N}_i} Q_s^T Q_s (\mathbf{x}_{sf,i} - \mathbf{x}_{sf,j}) = \beta \hat{\mathbf{d}}_{s,i} \tag{3.25}$$

where $\beta \in \mathbb{R}$ and the search direction $\hat{\mathbf{d}}_{s,i}$ represents the subspace reachable surface unit normal vector. By decomposing the final subspace states in components perpendicular and within the tangent space to the surface at $\mathbf{x}_{sf,i}$, denoted by $T_{\mathbf{x}_{sf}} M$

$$\begin{aligned} \sum_{j \in \mathcal{N}_i} [Q_s^T Q_s \mathbf{x}_{sf,i}]_\perp - [Q_s^T Q_s \mathbf{x}_{sf,j}]_\perp &= \beta \hat{\mathbf{d}}_{s,i} \\ \sum_{j \in \mathcal{N}_i} [Q_s^T Q_s \mathbf{x}_{sf,i}]_\parallel - [Q_s^T Q_s \mathbf{x}_{sf,j}]_\parallel &= \mathbf{0} \end{aligned} \tag{3.26}$$

where $(\cdot)_\perp$ represents the projection of a vector along the surface normal $\hat{\mathbf{d}}_s$ and $(\cdot)_\parallel$ denotes the remaining vector components that lie in the tangent space to the surface. From the

vector components that lie in $T_{\mathbf{x}_{sf,i}}M$,

$$\begin{aligned} \deg(i)[Q_s^T Q_s \mathbf{x}_{sf,i}]_{\|} &= \sum_{j \in \mathcal{N}_i} [Q_s^T Q_s \mathbf{x}_{sf,j}]_{\|} \\ [Q_s^T Q_s \mathbf{x}_{sf,i}]_{\|}^* &= \frac{1}{\deg(i)} \sum_{j \in \mathcal{N}_i} [Q_s^T Q_s \mathbf{x}_{sf,j}]_{\|} \end{aligned} \tag{3.27}$$

This result shows that equilibrium condition for the subspace states is for each weighted point solution to be the average of all of its weighted neighbors within $T_{\mathbf{x}_{sf,i}}M$. Expressed differently, each sample on the surface is the center of mass of the distribution of its neighboring particles in the local tangent plane. When this condition is applied to every point solution in the subspace reachable set, the result is uniform subspace point solution sampling in the tangent spaces of the surface.

□

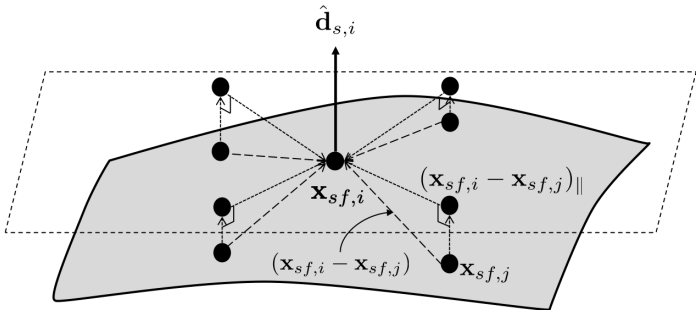


Figure 3.2: Illustration of Laplacian potential equilibirum condition

This energy minimization technique described in Lemma 2 also mirrors heat diffusion described by the heat equation. In this case, the continuous Laplacian operator is replaced by the graph Laplacian from graph theory [40]. Intuitively, given an uneven initial distribution of heat, the heat will diffuse and evenly spread across nodes as time progresses. The steady state value of heat will then be the average of all the initial heat values at all the nodes. In this application, the Laplacian cost quantifies the density of samples along a surface. In a similar manner to the heat diffusing through a series of nodes to an uniform, equal heat, the local and global density of particles will equalize to a uniform distribution.

In the field of structural mechanics, finite element methods use a similar concept. In finite element methods, a continuous body is modeled using discrete, interconnected nodes. The minimum total potential energy principle is used in many cases to solve for mechanical properties at the nodes [98].

Because the point solution final states are constrained to lie on the subspace reachable set, motion of the point solutions on this set boundary is also constrained to this surface. To avoid techniques that require Newton-Raphson projections or nonlinear equation solving [35, 34], the search directions $\hat{\mathbf{d}}_s(\boldsymbol{\theta})$ that parametrize the reachable subspace sampling are varied instead of the final states. As previously discussed, a sample of the subspace reachable set boundary is acquired by solving the optimal control problem in Eq. (2.3) parametrized by $\boldsymbol{\theta}$. Consequently, it is possible to adjust the mesh state through $\tilde{\boldsymbol{\theta}}$ to achieve the uniform reachable set surface sampling.

To reach the minimum Laplacian cost given an initial non-uniform distribution of point solutions, the following optimization problem needs to be solved

$$\min_{\tilde{\boldsymbol{\theta}}} J(\tilde{\boldsymbol{\theta}}) \tag{3.28}$$

Definition 5: Trajectory Flow State

Define the trajectory flow state as the concatenation between point solution state and costate along an optimal trajectory

$$\mathbf{y} = \begin{bmatrix} \mathbf{x}^T & \mathbf{p}^T \end{bmatrix}^T \tag{3.29}$$

It is then possible to analytically express the variation in the Laplacian cost function with changes in the search directions.

$$\frac{\partial J}{\partial \tilde{\boldsymbol{\theta}}} = \frac{\partial J}{\partial \tilde{\mathbf{y}}_f} \frac{\partial \tilde{\mathbf{y}}_f}{\partial \tilde{\mathbf{y}}_0} \frac{\partial \tilde{\mathbf{y}}_0}{\partial \tilde{\mathbf{z}}} \frac{\partial \tilde{\mathbf{z}}}{\partial \tilde{\boldsymbol{\theta}}} \tag{3.30}$$

$$\frac{\partial J}{\partial \tilde{\mathbf{y}}_f} = 2[\tilde{\mathbf{x}}_f^T L \quad \mathbf{0}_{1\times np}] \tag{3.31}$$

$$\frac{\partial \tilde{\mathbf{y}}_f}{\partial \tilde{\mathbf{y}}_{0\,i}} = \Phi_{y,i}(T, t_0) \tag{3.32}$$

$$\frac{\partial \tilde{\mathbf{y}}_0}{\partial \tilde{\mathbf{z}}_{\ i}} = \begin{bmatrix} \mathbb{I}_n & \mathbf{0}_{n\times 1} \\ -\frac{\partial^2 g}{\partial \mathbf{x}_0^2}[\mathbf{0}_{1\times n} \;\; 1]\mathbf{z}_i & -\frac{\partial g}{\partial \mathbf{x}_0}^T \end{bmatrix} \tag{3.33}$$

$$\frac{\partial \tilde{\mathbf{z}}}{\partial \tilde{\boldsymbol{\theta}}_i} = -\left[\frac{\partial \mathbf{F}_i}{\partial \mathbf{z}_i}\right]^{-1} \frac{\partial \mathbf{F}_i}{\partial \boldsymbol{\theta}_i} \tag{3.34}$$

where $(\cdot)_i$ denotes the i_{th} block matrix along the diagonal of the overall matrix. Since the mapping from $\boldsymbol{\theta}$ to $\mathbf{y}_f$ is independent from one point solution to another, these matrices are all block diagonal. The coupling between point solutions occurs through $\frac{\partial J}{\partial \tilde{\mathbf{y}}_f}$ and particularly through the Laplacian matrix L. The Laplacian matrix captures the graph structure by describing the neighbors of each point solution. However, because the Laplacian matrix only captures the relationships between a point solution and its neighbors, only local information from a point solution and its neighbors is required. This shows that the construction of this global Laplacian cost function and its gradient is indeed decentralized. In addition, Tsitsiklis et al. has shown that many asynchronous decentralized gradient optimization algorithms share convergence properties with their corresponding centralized implementation [99].

With the Laplacian cost function and its gradient defined, there are a number of methods to achieve a uniformally sampled surface boundary by solving Eq. (3.28).

Proposition 3: Continuous time gradient descent uniform coverage controller

One can form a continuous time dynamic system that describes the motion of the search directions angles, $\tilde{\boldsymbol{\theta}}$. As aforementioned, it is possible to subject the particles to the negative

gradient flow of the potenial function in order to reach a minimum corresponding to a uniform surface sampling. The resulting system is written as

$$\dot{\tilde{\boldsymbol{\theta}}} = \frac{d\tilde{\boldsymbol{\theta}}}{d\tilde{t}} = \tilde{\mathbf{u}} = -\frac{\partial J^T}{\partial \tilde{\boldsymbol{\theta}}} \tag{3.35}$$

where $\tilde{t}$ denotes the redistribution time, a separate independent variable used for the surface coverage system. Furthermore, this system is asymptotically stable.

Additionally, to avoid recomputing the entire reachable set, the optimal initial conditions can also be simultaneously updated using

$$\frac{d\tilde{\mathbf{z}}}{d\tilde{t}} = \frac{\partial \tilde{\mathbf{z}}}{\partial \tilde{\boldsymbol{\theta}}} \frac{d\tilde{\boldsymbol{\theta}}}{d\tilde{t}} \tag{3.36}$$

Proof: To demonstrate stability, the Lyapunov function is defined as

$$\bar{V}(\tilde{\boldsymbol{\theta}}_e) = J(\tilde{\boldsymbol{\theta}}_e) \tag{3.37}$$

where $\tilde{\boldsymbol{\theta}}_e = \tilde{\boldsymbol{\theta}} - \tilde{\boldsymbol{\theta}}^*$ and $\tilde{\boldsymbol{\theta}}^*$ is the minimizing mesh angle state. From Lemma 2,

$$\frac{\partial^2 \bar{V}}{\partial \tilde{\boldsymbol{\theta}}_e^2} = \frac{\partial^2 J}{\partial \tilde{\boldsymbol{\theta}}_e^2} = Q_s^T Q_s > 0$$

Consequently when considering feasible mesh state variations near the minimizer,

$$\bar{V}(\tilde{\boldsymbol{\theta}}_e) = 0 \text{ iff } \tilde{\boldsymbol{\theta}}_e = 0$$
$$\bar{V}(\tilde{\boldsymbol{\theta}}_e) > 0 \text{ iff } \tilde{\boldsymbol{\theta}}_e \neq 0$$

making the Laplacian cost potential energy a candidate Lyapunov function. Evaluating the

redistribution time derivative of $\bar{V}$ for the system in Eq. (3.35),

$$\frac{d\bar{V}}{d\tilde{t}} = \frac{dJ}{d\tilde{t}} = \frac{\partial J}{\partial \tilde{\boldsymbol{\theta}}} \frac{d\tilde{\boldsymbol{\theta}}}{d\tilde{t}} = -\left\| \frac{\partial J}{\partial \tilde{\boldsymbol{\theta}}} \right\|^2 < 0$$

Using Lyapunov theory provides the sufficient condition for an asymptotically stable system that converges to the potential energy minimum given enough redistribution time, $\tilde{t}$.

□

This system in Eq. (3.35) and Eq. (3.36) may encounter numerical issues related with the stiffness of the differential equation in Eq. (3.35) and the possible singularity of $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$. To help alleviate the issues due to the singularity of the jacobian $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$, one can use a pseudo-arclength equivalent to Eq. (3.35) and Eq. (3.36). Additionally, the negative gradient descent may lead to slow convergence depending on the number and initial distribution of the particles.

Proposition 4: Discrete gradient-descent redistribution

As opposed to a continuous time dynamical system, one can form a gradient descent method by iteratively updating the mesh angle state.

$$\tilde{\boldsymbol{\theta}}_{k+1} = \tilde{\boldsymbol{\theta}}_k - \gamma_k D_k \frac{\partial J}{\partial \tilde{\boldsymbol{\theta}}}\bigg|_{\tilde{\boldsymbol{\theta}}_k} \tag{3.38}$$

where γ_k is the step size and D_k is a positive definite symmetric matrix, both at iteration k of an iterative descent algorithm [100, 43].

□

If the gradient descent algorithm is implemented as written in Eq. (3.38), the each point solution sample of the subspace reachable set would have to be computed using the methodology outlined in Chapter 2. With every $\boldsymbol{\theta}_i$, $\mathbf{z}_i(T=0)$ is recomputed and Eq. (2.32) is used to propagate $\mathbf{z}_i(T)$. This process can easily become tedious as all the samples of the subspace reachable set are propagated over time, quickly increasing the required computational time

to achieve uniform surface coverage.

However, an alternate approach to implementing Eq. (3.38) relies on numerical continuation. It is possible to fix the time horizon T and choose a directional parameter to explore nearby extremal solutions on the reachable set boundary. By choosing the vector of angular coordinates, $\boldsymbol{\theta}$, as the continuation parameter, the continuation in Eq. (2.32) may be rewritten as

$$\frac{d\mathbf{z}}{d\boldsymbol{\theta}} = -\left[\frac{\partial \mathbf{F}}{\partial \mathbf{z}}\right]^{-1} \frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}} \tag{3.39}$$

The continuation method in Eq. (3.39) represents the differential change in the optimal control problem solution given a change in $\hat{\mathbf{d}}_s(\boldsymbol{\theta})$. The Jacobian matrix $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$ is unchanged from Eq. (2.32) but $\frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}}$ must be computed as follows

$$\frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial}{\partial \hat{\mathbf{d}}_s}(\hat{\mathbf{d}}_s \hat{\mathbf{d}}_s^T \mathbf{x}_f) \\ \mathbf{0}_{1\times n-1} \end{bmatrix} = \begin{bmatrix} (\hat{\mathbf{d}}_s^T \mathbf{x}_f \mathbb{I}_n + \hat{\mathbf{d}}_s \mathbf{x}_f^T) \frac{\partial \hat{\mathbf{d}}_s}{\partial \boldsymbol{\theta}} \\ \mathbf{0}_{1\times n-1} \end{bmatrix} \tag{3.40}$$

where $\frac{\partial \hat{\mathbf{d}}_s}{\partial \boldsymbol{\theta}}$ shows the relationship between angular parameter and subspace unit vector. If hyperspherical coordinates are used to parametrize $\hat{\mathbf{d}}_s$, this term can be computed analytically.

To reparametrize Eq. (3.39) as a function of a scalar independent parameter η, it is possible to use the following linear interpolation

$$\boldsymbol{\theta}(\eta) = \boldsymbol{\theta}_0(1-\eta) + \boldsymbol{\theta}_f \eta, \eta \in [0,1] \tag{3.41}$$

where $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_f$ denote the initial and final, target $\boldsymbol{\theta}$ in the numerical condition. The

updated form for Eq. (3.39) is

$$\begin{aligned}\frac{d\mathbf{z}}{d\eta} &= -\left[\frac{\partial \mathbf{F}}{\partial \mathbf{z}}\right]^{-1}\frac{\partial \mathbf{F}}{\partial \eta} \\ &= -\left[\frac{\partial \mathbf{F}}{\partial \mathbf{z}}\right]^{-1}\frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{\theta}}{\partial \eta} \\ &= -\left[\frac{\partial \mathbf{F}}{\partial \mathbf{z}}\right]^{-1}\frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_f - \boldsymbol{\theta}_0)\end{aligned} \tag{3.42}$$

where $\eta \in [0, 1]$. In the case of the iterative gradient descent described in Eq. (3.38), $\boldsymbol{\theta}_0 = \boldsymbol{\theta}_k$ and $\boldsymbol{\theta}_f = \boldsymbol{\theta}_{k+1}$.

As this is for an unconstrained nonlinear optimization problem, careful considerations must be taken when choosing the descent direction (specified by D_k) and the step size. Choice of descent direction could lead to very slow convergence. While a large step size may not violate domain constraints for angular coordinates, these steps may disrupt the graph communication structure by changing the nearest neighbors of particles. While there are techniques to reevaluate the nearest neighbors of a collection of particles such as using k-d trees [101], the aim of this work is to outline techniques that maintain the graph structure throughout the optimization.

In order to ensure a sufficient decrease in the Laplacian cost without small step sizes, the Wolfe conditions are often used [43, 100]. Combined, these conditions outline a range of acceptable step sizes that seek to minimize the Laplacian cost along the descent direction.

Proposition 5: (Theorem 4.1 [102], Proof in paper), Asymptotic stability of line-search descent algorithm with Wolfe Conditions

Consider a line search descent algorithm that terminates if $\frac{\partial J}{\partial \tilde{\boldsymbol{\theta}}}^T = \mathbf{0}$. If the search direction is a descent direction, the step size satisfies both the Wolfe conditions, and the cost function

is analytic, then there exists a single point $\tilde{\boldsymbol{\theta}}^*$ such that

$$\lim_{k\to\infty} \tilde{\boldsymbol{\theta}}_k = \tilde{\boldsymbol{\theta}}^*$$

where $\tilde{\boldsymbol{\theta}}^*$ is the stationary point of J so $\frac{\partial J}{\partial \tilde{\boldsymbol{\theta}}}^T(\tilde{\boldsymbol{\theta}}^*) = \mathbf{0}$ [102].

This result from Proposition 5 proves that the iterative descent algorithm in Eq. (3.38) will asymptotically converge to the Laplacian cost minimum corresponding to the uniform distribution of samples on the surface boundary. For faster convergence, one can use descent directions techniques such as diagonally-scaled steepest descent, Newton's method, BFGS, and Quasi-Newton's methods [43, 100, 34] as higher-order information is used or approximated.

3.2.2 Spawning or Deleting Samples

One technique of equilibriating the distances between neighboring point solutions is to add new or remove old point solutions in the graph.

Definition 6: Bounded point-wise distance threshold

For two neighboring point solutions, a point-wise distance metric can be defined using the states on the reachable set boundary, $d(\mathbf{x}_f(\mathbf{z}_i), \mathbf{x}_f(\mathbf{z}_j))$ where $\mathbf{z}_i$ denotes the optimal control problem solution from Eq. (2.29) of the i_{th} point solution. Feasible point-wise distances between neighboring point solutions can be defined using lower and upper bounds on the distance metric

$$0 \leq d_l \leq d(\mathbf{x}_f(\mathbf{z}_i), \mathbf{x}_f(\mathbf{z}_j)) \leq d_u \tag{3.43}$$

where $d_u - d_l > 0$. It is not required for $d(\mathbf{x}_f(\mathbf{z}_i), \mathbf{x}_f(\mathbf{z}_j))$ to be a metric, but the properties of metrics may be useful for nearest-neighbor search applications.

By prescribing lower and upper bounds for this distance metric, detecting sparse or dense

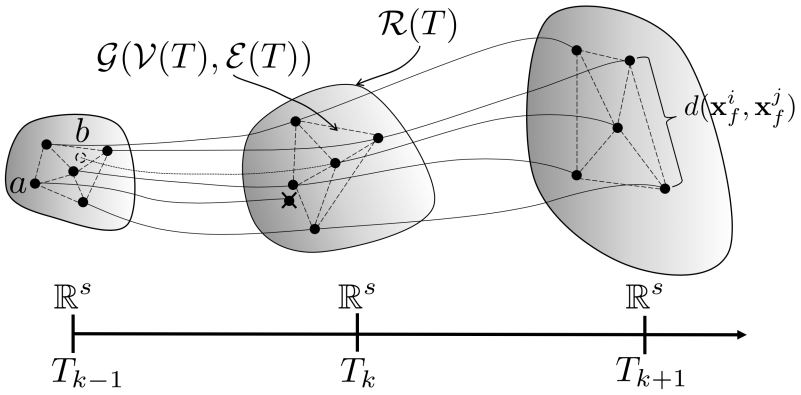


Figure 3.3: Visualization of the propagation of the graph $\mathcal{G}$ with deletion and insertion of point solutions on the subspace reachable set boundary in $\mathbb{R}^s$ denoted by points a and b, respectively

regions on the extremal surface could be performed by identifying point solution pairs that do not satisfy the given bounds. As the reachable set evolves in time, it is then possible to spawn new point solutions when $d(\mathbf{x}_f(\mathbf{z}_i), \mathbf{x}_f(\mathbf{z}_j)) > d_u$ and remove redundant point solutions when $d(\mathbf{x}_f(\mathbf{z}_i), \mathbf{x}_f(\mathbf{z}_j)) < d_l$. This process can be visualized using Fig.3.3 where the point solution denoted by a was removed and the point solution denoted by b was spawned. Either of these changes may occur independently and will each result in a updated graph $\mathcal{G}$.

While there are many methods of determining whether or not an edge connecting multiple point solutions on the reachable set boundary is too large, this work examines the use of univariate statistics. Because an approximate uniform distribution of point solutions is sought, the ideal distribution of point-wise distances between neighboring point solutions should have a small sample variance. One efficient method to reduce the sample variance through the deletion of samples is by identifying outliers. After computing the point-wise edge distance between each neighboring point solutions on the reachable set boundary, one can attempt to identify outliers of the resulting distribution. Furthermore, only the outliers corresponding to larger point-wise distances are considered for the creation of new point solutions.

One fairly simple and robust method for labelling outliers drawn from an unknown probability distribution is the interquartile range (IQR) method [103]. For a univariate distribution, one can compute the Q1, Q2, and Q3 quartiles corresponding to different percentiles, typically chosen as the 25th, 50th, and 75th percentiles, respectively. In general, qth percentiles are values in the distribution below which $q\%$ of the samples in the distribution fall. The IQR value is defined as Q3 - Q1 and is a measure of the spread of the central collection of samples in the distribution. By convention, a sample is labelled an outlier if it is greater than Q3 + 1.5IQR or less than Q1 - 1.5IQR. IQR is a more robust measure of the dispersion of data than the conventional measures of dispersion, such as sample variance, as the latter is greatly influenced by outliers. If the underlying probability distribution is normal, this outlier threshold corresponds to 2.698σ [103].

Once the point-wise distance outliers are detected, new point solutions can be spawned from neighbors by bisecting their search directions. If the edge distance from point solution i to point solution j was deemed too large, then a new bisecting search direction can be computed

$$\begin{aligned} \hat{\mathbf{d}}_{s,new} &= \frac{\hat{\mathbf{d}}_s(\boldsymbol{\theta}_i) \; + \; \hat{\mathbf{d}}_s(\boldsymbol{\theta}_j)}{\|\hat{\mathbf{d}}_s(\boldsymbol{\theta}_i) \; + \; \hat{\mathbf{d}}_s(\boldsymbol{\theta}_j)\|} \\ \boldsymbol{\theta}_{new} &= \mathbf{h}^{-1}(\hat{\mathbf{d}}_{s,new}) \end{aligned} \tag{3.44}$$

The transformation from search direction unit vector to the vector of angle coordinates in Eq. (3.44) is not unique in that there can be an infinite number of $\boldsymbol{\theta}$ vectors corresponding to the same $\hat{\mathbf{d}}_s$ vector. As a result, operations between point solutions should be performed on the search direction unit vectors as opposed the vector of angular coordinates, if possible.

Another technique for spawning particles relies on being able to choose new particles that minimize the global potential function cost impact when they are spawned. Suppose there is a pair of existing particles, denoted particle 1 and particle 2, with a relatively large weighted distance between them in the subspace of interest. If a new particle, denoted by particle γ, were to be spawned between them, the potential energy cost associated with this

new addition is given by

$$J_{new} = (\mathbf{x}_{sf,1} - \mathbf{x}_{sf,\gamma})^T Q_s^T Q_s (\mathbf{x}_{sf,1} - \mathbf{x}_{sf,\gamma}) + (\mathbf{x}_{sf,2} - \mathbf{x}_{sf,\gamma})^T Q_s^T Q_s (\mathbf{x}_{sf,2} - \mathbf{x}_{sf,\gamma}) \tag{3.45}$$

In addition, suppose this new particle was parametrized as a spherical interpolation (slerp) of the existing particles' search directions such as

$$\hat{\mathbf{d}}_{s,\gamma} = \frac{\sin[(1-\gamma)\Omega]}{\sin\Omega}\,\hat{\mathbf{d}}_{s,1} + \frac{\sin[\gamma\Omega]}{\sin\Omega}\,\hat{\mathbf{d}}_{s,2} \tag{3.46}$$

where $\gamma \in [0,1]$ and Ω denotes the angle between $\hat{\mathbf{d}}_{s,1}$ and $\hat{\mathbf{d}}_{s,2}$ [104]. Spherical interpolation of this form parametrizes a hyperspherical arc between the given unit vectors.

The objective is now to minimize $J_{new}(\gamma)$ to minimize the global potential energy generated by spawning this new particle. As there are four pieces of information available about this $J_{new}(\gamma)$ curve, a cubic polynomial is created. The four pieces of information are described below

$$\begin{aligned}
J_{new}(\gamma=0) &= (\mathbf{x}_{sf,2} - \mathbf{x}_{sf,1})^T Q_s^T Q_s (\mathbf{x}_{sf,2} - \mathbf{x}_{sf,1}) = J_{1,2} \\
J_{new}(\gamma=1) &= (\mathbf{x}_{sf,1} - \mathbf{x}_{sf,2})^T Q_s^T Q_s (\mathbf{x}_{sf,1} - \mathbf{x}_{sf,2}) = J_{1,2} \\
\frac{dJ_{new}}{d\gamma}(\gamma=0) &= \frac{\partial J_{new}}{\partial \hat{\mathbf{d}}_s}\frac{\partial \hat{\mathbf{d}}_s}{\partial \gamma}\bigg|_{\gamma=0} = J_1^{'} \\
\frac{dJ_{new}}{d\gamma}(\gamma=1) &= \frac{\partial J_{new}}{\partial \hat{\mathbf{d}}_s}\frac{\partial \hat{\mathbf{d}}_s}{\partial \gamma}\bigg|_{\gamma=1} = J_2^{'}
\end{aligned} \tag{3.47}$$

With the four quantities computed, the minimum of the cubic polynomial is computed as

$$\gamma^* = \begin{cases} \frac{1}{2} & \text{if } J_1^{'} = -J_2^{'} \\ \frac{2J_1^{'} + J_2^{'} \pm \sqrt{J_1^{'2} + J_1^{'}J_2^{'} + J_2^{'2}}}{3(J_1^{'} + J_2^{'})} & \text{otherwise} \end{cases} \tag{3.48}$$

where γ^* is enforced to be between 0 and 1.

Altogether, either the search vector bisection or the cubic minimization mesh refinement

process can be repeated any number of times to achieve a desired surface resolution or until a specified number of maximum refinement iterations are completed.

It is possible to delete point solutions from the reachable set boundary that are sufficiently close to neighbors. This procedure reduces the overall computational load required as very close point solutions, with redundant information describing the boundary of the reachable set, would not be propagated through the full continuation method. Similar logic using the IQR method could be used to prune those point solutions from further propagation. However, this case is not considered in this thesis.

3.3 Results

To demonstrate the subspace reachable surface sampling, this example examines the forward subspace reachable set of a 3-DOF coupled Duffing oscillator. The nonlinear equations of motion are

$$\begin{aligned} m_1\ddot{x}_1 = -k_{1,1}x_1 - k_{1,3}x_1^3 - f_1\dot{x}_1 \\ +k_{2,1}(x_2 - x_1) + k_{2,3}(x_2 - x_1)^3 + f_2(\dot{x}_2 - \dot{x}_1) \end{aligned} \tag{3.49}$$

$$\begin{aligned} m_2\ddot{x}_2 = \\ -k_{2,1}(x_2 - x_1) - k_{2,3}(x_2 - x_1)^3 - f_2(\dot{x}_2 - \dot{x}_1) \\ +k_{3,1}(x_3 - x_2) + k_{3,3}(x_3 - x_2)^3 + f_3(\dot{x}_3 - \dot{x}_2) \end{aligned} \tag{3.50}$$

$$\begin{aligned} m_3\ddot{x}_3 = -k_{3,1}(x_3 - x_2) - k_{3,3}(x_3 - x_2)^3 \\ -f_3(\dot{x}_3 - \dot{x}_2) + u \end{aligned} \tag{3.51}$$

where m_i represent the masses, x_i represent the positions, $\dot{x}_i$ are the velocities, $k_{i,1}$ represent the linear spring coefficients, $k_{i,3}$ represent the cubic spring coefficients, f_i represent viscous friction coefficients, and u is forcing term acting only on m_3 with a maximum force

of u_m. For this demonstration, the following parameters are chosen:

$$g(\mathbf{x}_0) = \mathbf{x}_0^T\mathbf{x}_0 - 1 = 0$$

$$m_1 = m_2 = m_3 = 1 \quad k_{1,1} = k_{2,1} = k_{3,1} = 1$$

$$k_{1,3} = k_{2,3} = k_{3,3} = 1/9 \quad f_1 = f_2 = f_3 = 1$$

$$u_m = 1, T_f = \pi$$

3.3.1 Search Angle Bisection using IQR

To compute the subspace reachable set, $\theta \in [0, 2\pi)$ is evenly sampled 30 times. For each θ, the corresponding search direction vector $\hat{\mathbf{d}}_s$ was computed and the continuation method outlined in Eq. (2.32) was performed to generate point solutions on the forward reachable set. The point solutions are clustered towards the high-curvature regions of the reachable set boundary.

After computing the forward reachable set described by these 30 point solutions, the cumulative distribution function of edge distances between neighboring points are displayed in Figure 3.5. If one desires a more uniform description of the reachable set boundary, it is possible to achieve a more uniform distribution by spawning additional point solutions. The IQR method identifies edges that would need bisecting point solutions spawned. After the mesh refinement, the edge distance distribution is less multimodal and more uniform than it was originally, resulting in a cumulative distribution function with a steeper slope. The additional samples can be seen in Figure 3.4 denoted by the black markers. This computation was completed in about 15 seconds on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor with 8 GB 2133 MHz RAM.

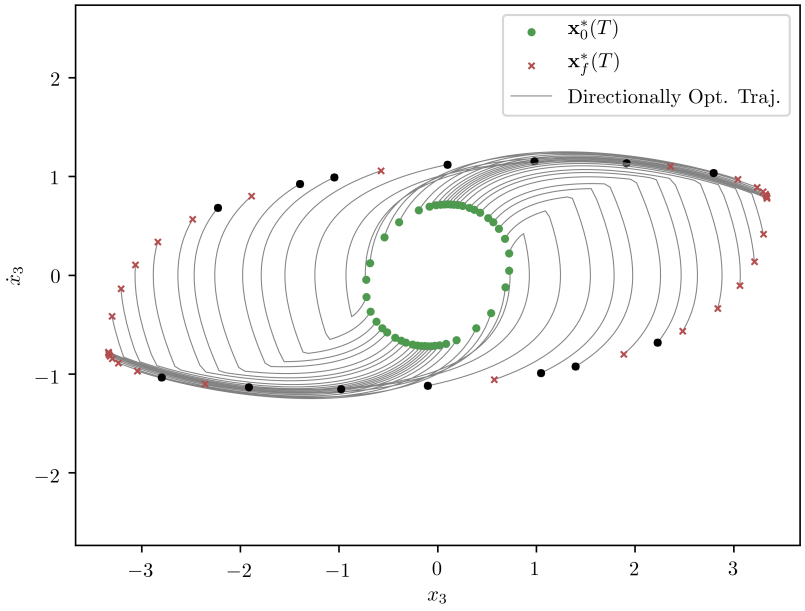


Figure 3.4: Forward x_3 subspace reachable set point solution trajectories at $T = \pi$ for 6-dimensional nonlinear Duffing oscillator with IQR-based sample insertion where black markers denote inserted samples

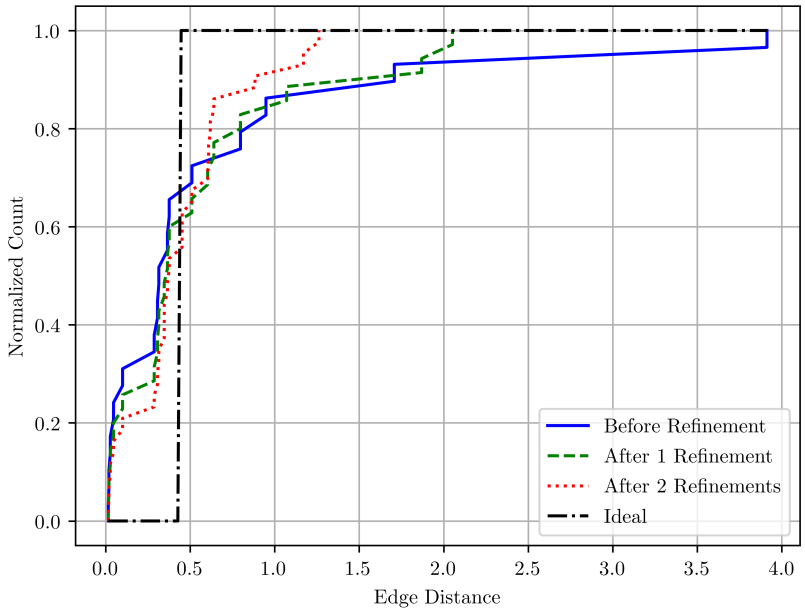


Figure 3.5: Point-wise edge distance cumulative distribution function for 6-dimensional nonlinear Duffing oscillator forward x_3 subspace reachable set before and after mesh refinement process using IQR outlier bisection

3.3.2 Potential Function Gradient Descent Redistribution

As previously discussed, it is also possible to achieve more uniform surface sampling by redistributing the point solutions themselves using Lyapunov theory and energy minimization concepts.

In this example, the overall reachablility time horizon is subdivided into subintervals delimited by the intermediate, redistribution times T_s. At these redistribution times, the time evolution of the reachable set is frozen while the point solutions are redistributed by changing $\tilde{\boldsymbol{\theta}}$ using any of the techniques outlined above. In this demonstration, the chosen redistribution times are $T_{s,1} = T_f/2 = \pi/2, T_{s,2} = T_f = \pi$.

For this demonstration, the Lyapunov controller is implemented and the results are shown. For each of the following figures, each row of the plots correspond to the redistribution results at each of the times T_s. Figure 3.6 displays the subspace reachable set point solutions at each of the redistribution times. For the initial redistribution time $T_{s,0} = 0$, the initial sampling of the subspace reachable set is uniform because the initial condition constraint surface is spherical. Figure 3.7 demonstrates the increase in surface sampling uniformity by looking at the point-wise edge distance distributions before and after the redistributions are performed.

In this example, the final distribution of point solutions on the reachable set is more uniform than the curvature-based sampling. In Figure 3.6, there are cusps in the forward reachable set boundary for this particular system. With these cusps, the reachable set boundary is not $\mathcal{C}^1$ and the continuation method Jacobian $\frac{\partial \mathbf{F}}{\partial \mathbf{z}}$ approaches singularity as a point solution approaches the cusps. When using the continuous time Lyapunov controller, the cusps in the reachable set boundary lead to ill-conditioned, stiff ODEs. These generally require longer computation time and suffer decreased solution accuracy. Because of this, the computation took about 3 hours on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor

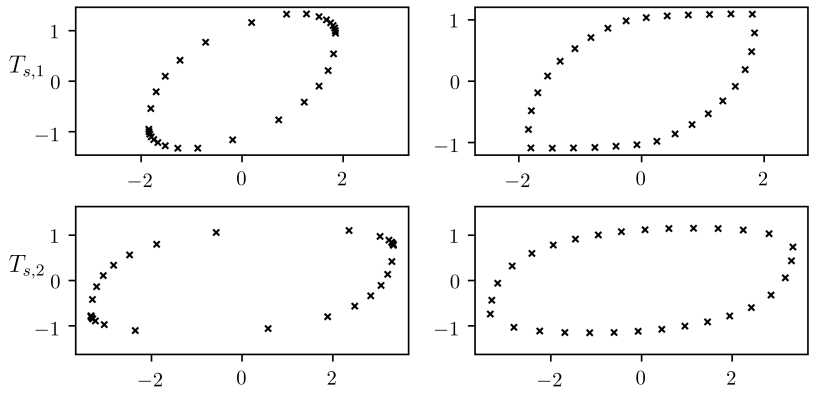


Figure 3.6: Forward x_3 subspace reachable set sampling for $T_s = T_f/2, T_s = T_f$ for 6-dimensional nonlinear Duffing oscillator both before (left) and after (right) sample redistribution

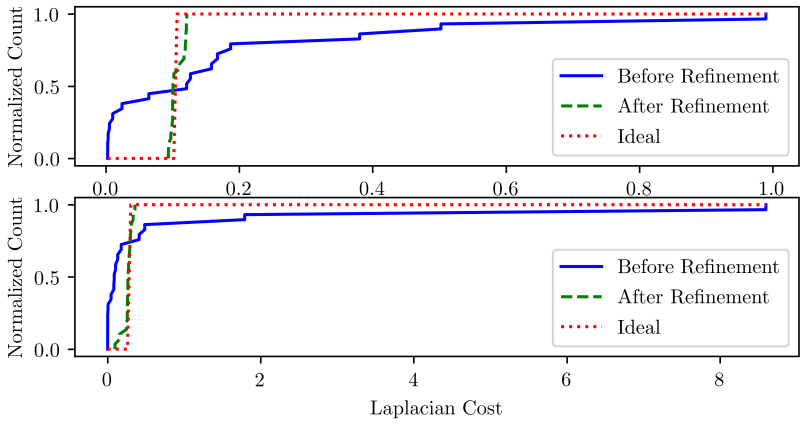


Figure 3.7: Point-wise edge distance cumulative distribution function for 6-dimensional nonlinear Duffing oscillator forward x_3 subspace reachable set before and after mesh refinement process using gradient descent for $T_s = T_f/2, T_s = T_f$

with 8 GB 2133 MHz RAM. While not implemented in this chapter, drastic speed improvements can be made by using gradient-based line search optimization techniques with Wolfe conditions [100].

3.4 Conclusions

This chapter discusses the use of parameterized optimal control and continuation methods to sample points on the convex hull of a subspace reachable set. Furthermore, by specifying a uniform distribution of initial search directions, this chapter shows that the resulting

reachable set sampling clusters towards high curvature regions of the surface boundary.

Multiple techniques for updating the distribution of point solutions on the reachable set boundary are discussed. When a particular reachable set resolution is specified, additional point solutions can be spawned to fill in gaps in the surface boundary. Alternatively, when computational memory is of significance, the number of point solutions may remain fixed. In this case, multiple decentralized techniques are presented to update the distribution of point solutions on the reachable set boundary.

A comparison with two other commonly referenced reachable set analysis tools was also performed. It is shown that the presented methodology can provide accurate samples of the reachable set boundary without incurring a large computational cost. While the toolbox comparison shown here is not comprehensive, it highlights some advantages of the presented methodology such as the computational efficiency of continuation-based reachability, the accuracy measures provided by the necessary conditions of optimality in optimal control, and the use of dimensionally scalable support functions for representations of convex sets.

CHAPTER 4

CONNECTIONS WITH REACHABILITY THEORY AND MULTI-OBJECTIVE OPTIMIZATION

4.1 Reachability optimal control formulation

An optimal reachability problem can be defined as a continuum of optimal control problems with prescribed initial conditions. The optimal control problem is formally stated as

$$\begin{aligned} \max_{\mathbf{u}\in U}\{J\} = \max_{\mathbf{u}\in U}\Bigg[\int_{t_0}^{t_f} \mathcal{L}(\mathbf{x}(\tau),\mathbf{u}(\tau),\tau)d\tau + V(\mathbf{x}_f,t_f)\Bigg] \\ s.t. \qquad \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{u},t) \\ \mathbf{g}(\mathbf{x}_0) = \mathbf{0} \end{aligned} \tag{4.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state, $\mathbf{u} \in \mathbb{R}^m$ is the control input, $t \in [t_0, t_f]$ is time, $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ is the trajectory performance function, $V : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is the terminal performance function, $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n$ captures the system dynamics, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^v$ expresses initial conditions, and $U \subseteq \mathbb{R}^m$ defines the set of admissible controls. Inequality constraints on the initial states can be formed by introducing slack variables [42]. However, for reachability optimal control problems, an equality constraint is sufficient as it describes the boundary of the reachable set.

To account for the static and dynamic constraints, the augmented performance index is defined

$$\tilde{J}(\mathbf{u}) = \int_{t_0}^{t_f} \left[\mathcal{L}(\mathbf{x}(\tau),\mathbf{u}(\tau),\tau) + \mathbf{p}^T(\mathbf{f}(\mathbf{x},\mathbf{u},\tau) - \dot{\mathbf{x}})\right] d\tau + V(\mathbf{x}_f,t_f) + \boldsymbol{\lambda}^T\mathbf{g}(\mathbf{x}_0) \tag{4.2}$$

where $\mathbf{p} \in \mathbb{R}^n$, the Lagrange multiplier for the state dynamics, is denoted as the costate variable and $\boldsymbol{\lambda} \in \mathbb{R}^v$ is a Lagrange multiplier for the initial condition constraint.

Define the optimal control Hamiltonian as

$$\mathcal{H}^*(\mathbf{x}, \mathbf{p}, \mathbf{u}, t) = \max_{\mathbf{u} \in U} \left[\mathcal{L}(\mathbf{x}, \mathbf{u}, t) + \mathbf{p}^T \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \right] \tag{4.3}$$

From calculus of variations [105], the first-order necessary conditions of optimality are as follows

$$\dot{\mathbf{x}} = \frac{\partial \mathcal{H}^{*T}}{\partial \mathbf{p}} \tag{4.4a}$$

$$\dot{\mathbf{p}} = -\frac{\partial \mathcal{H}^{*T}}{\partial \mathbf{x}} \tag{4.4b}$$

$$\mathbf{u} = \underset{\mathbf{u} \in U}{\text{argmax}} \left\{ \mathcal{H} \right\} \tag{4.4c}$$

$$\mathbf{p}_f = \frac{\partial V}{\partial \mathbf{x}_f}^T \tag{4.4d}$$

$$\mathbf{p}_0 = -\frac{\partial \mathbf{g}}{\partial \mathbf{x}_0}^T \boldsymbol{\lambda} \tag{4.4e}$$

$$\mathbf{g}(\mathbf{x}_0) = \mathbf{0} \tag{4.4f}$$

From Eq. (4.4e), note that the initial costate $\mathbf{p}_0$ is solely a function of $\mathbf{x}_0$ and $\boldsymbol{\lambda}$. Expressing the terminal value of the state and costate trajectories using flow functions

$$\mathbf{x}_f = \mathbf{x}(t_f) = \phi_x(t_f; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \boldsymbol{\lambda})) \tag{4.5a}$$

$$\mathbf{p}_f = \mathbf{p}(t_f) = \phi_p(t_f; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \boldsymbol{\lambda})) \tag{4.5b}$$

These expressions can be used to simplify the necessary conditions of optimality to

$$\phi_p(t_f; \mathbf{x}_0, \mathbf{p}_0\big(\mathbf{x}_0, \boldsymbol{\lambda}\big)) = \frac{\partial V}{\partial \mathbf{x}_f}^T \Big(\phi_x(t_f; \mathbf{x}_0, \mathbf{p}_0\big(\mathbf{x}_0, \boldsymbol{\lambda}\big))\Big) \tag{4.6a}$$

$$\mathbf{g}(\mathbf{x}_0) = \mathbf{0} \tag{4.6b}$$

where the free variables that need to be solved are $\mathbf{x}_0$ and $\boldsymbol{\lambda}$.

The necessary conditions of optimality may also be written in the form of a root-finding problem. By defining a function $\mathbf{F}_{OC} : \mathbb{R}^{n+v} \to \mathbb{R}^{n+v}$ where

$$\mathbf{F}_{OC}(\mathbf{x}_0, \boldsymbol{\lambda}) = \begin{bmatrix} \frac{\partial V}{\partial \mathbf{x}_f}^T \Big(\phi_x(t_f; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \boldsymbol{\lambda}))\Big) - \phi_p(t_f; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \boldsymbol{\lambda})) \\ \mathbf{g}(\mathbf{x}_0) \end{bmatrix} \tag{4.7}$$

and $\mathbf{z} \in \mathbb{R}^{n+v}$ denotes the optimal control problem solution vector defined as

$$\mathbf{z}_{OC} = \begin{bmatrix} \mathbf{x}_0 \\ \boldsymbol{\lambda} \end{bmatrix} \tag{4.8}$$

the following simple form is attained

$$\mathbf{F}_{OC}(\mathbf{z}_{OC}) = \mathbf{0} \tag{4.9}$$

This optimal control problem defined in Eq. (4.1) can be converted to a minimum-time reachability problem. In such case, the Lagrangian term may be set $\mathcal{L}(\mathbf{x}, \mathbf{u}, t) = 0$ [5]. Let the terminal performance function, $V(\mathbf{x}_f, t_f)$, be written as an inner product as

$$V(\mathbf{x}_f, t_f) = \hat{\mathbf{d}}_s^T \mathbf{x}_f \tag{4.10}$$

where $\hat{\mathbf{d}}_s \in \mathbb{S}^n$ is a search direction parameter and $\mathbb{S}$ denotes the unit hypersphere.

In this case, the first order conditions of optimality for the reachability problem are given by

$$\mathbf{F}_{OC}(\mathbf{x}_0, \boldsymbol{\lambda}; \hat{\mathbf{d}}_s) = \begin{bmatrix} \hat{\mathbf{d}}_s - \phi_p(t_f; \mathbf{x}_0, \mathbf{p}_0(\mathbf{x}_0, \boldsymbol{\lambda})) \\ \mathbf{g}(\mathbf{x}_0) \end{bmatrix} = \mathbf{0} \tag{4.11}$$

Brew et al. has shown that solving Eq. (4.11) for a given value of $\hat{\mathbf{d}}_s$ is equivalent to computing a sample of the reachable set boundary [63].

4.2 Multi-objective optimization Problem Formulation

Now we look at the necessary conditions for a multi-objective optimization problem. Let design variables be denoted as $\mathbf{d} \in \mathbb{R}^d$. Design variables are independently chosen variables that are chosen by the user and affect the objective functions. The set of feasible design variables is given by $D = \{\mathbf{d} \in \mathbb{R}^d : h_i(\mathbf{d}) = 0 \; \forall \; i = 1, \ldots, w\}$ where $\mathbf{h}(\mathbf{d}) = [h_1(\mathbf{d}), \ldots, h_w(\mathbf{d})]^T$ denotes the set of equality constraints. Problems with inequality constraints can be put into this form with the inclusion of slack variables [42]. Without loss of generality, this work only considers equality constraints. The set D is called the design space as it denotes the set of feasible design variables.

The objective functions are defined as

$$\mathbf{J} = \begin{cases} \mathbb{R}^d \to \mathbb{R}^p \\ \\ \mathbf{d} \to \mathbf{J}(\mathbf{d}) = [J_1(\mathbf{d}), \ldots, J_p(\mathbf{d})]^T \end{cases} \tag{4.12}$$

where the notation indicates that $\mathbf{J}$ is a mapping from $\mathbb{R}^d$ to $\mathbb{R}^p$ that maps a vector $\mathbf{d}$ to $\mathbf{J}(\mathbf{d})$ and is a collection of individual objective functions. The set of feasible objective functions is defined as $Y = \mathbf{J}[D] = \{\mathbf{J}(\mathbf{d}) : \mathbf{x} \in D\}$ and is called the objective space. The functions $\mathbf{J}$ and $\mathbf{h}$ are assumed twice continuously differentiable [55].

One of the fundamental results of multi-objective optimization came from Kuhn and Tucker

in the form of necessary conditions for Pareto optimality [58]. Given a multi-objective optimization problem of the form

$$\begin{aligned} \min_{\mathbf{d}} \quad & \mathbf{J}(\mathbf{d}) \\ s.t. \quad & \mathbf{h}(\mathbf{d}) = \mathbf{0} \end{aligned} \tag{4.13}$$

Let $\mathbf{d}^*$ be a Pareto optimal point of Eq. (4.13). Suppose that the set of vectors $\{\nabla h_i(\mathbf{d}) : i = 1, \ldots, w\}$ is linearly independent. Then there exists vectors $\boldsymbol{\mu} \in \mathbb{R}^w$ and $\alpha \in \mathbb{R}^p$ with $\alpha_i \geq 0, i = 1, \ldots, p$ and $\sum \alpha_i = 1$ such that

$$\sum_{i=1}^{p} \alpha_i \nabla J_i(\mathbf{d}^*) + \sum_{j=1}^{w} \mu_j \nabla h_j(\mathbf{d}^*) = 0 \tag{4.14a}$$

$$h_i(\mathbf{d}^*) = 0, i = 1, \ldots, w \tag{4.14b}$$

where $\boldsymbol{\alpha}$ quantifies relative weights between objective functions and $\boldsymbol{\mu}$ denotes the Lagrange multipliers for the equality constraint $\mathbf{h}$ [58]. The point $\mathbf{d}^*$ is called a critical point or a Karush-Kuhn-Tucker (KKT) point of the multi-objective problem given in Eq. (4.13).

Introducing the scalar-valued function

$$J_\alpha(\mathbf{d}) = \sum_{i=1}^{p} \alpha_i J_i(\mathbf{d}) = \boldsymbol{\alpha}^T \mathbf{J}(\mathbf{d}) \tag{4.15}$$

the first order necessary conditions given in Eq. (4.14) are equivalent to claiming that $\mathbf{d}^*$ is a KKT point to the single-objective optimization problem with the objective function $J_\alpha(\mathbf{d})$.

These necessary conditions of optimality given in Eq. (4.14) can be rewritten as a system of equations of the form

$$\mathbf{F}_{MO}(\mathbf{z}_{MO}) = \mathbf{0} \tag{4.16}$$

where

$$\mathbf{F}_{MO} = \begin{bmatrix} \sum_{i=1}^{p} \alpha_i \nabla J_i(\mathbf{d}) + \sum_{j=1}^{w} \mu_j \nabla h_j(\mathbf{d}) \\ \mathbf{h}(\mathbf{d}) \end{bmatrix}, \mathbf{z}_{MO} = \begin{bmatrix} \mathbf{d} \\ \boldsymbol{\mu} \end{bmatrix} \tag{4.17}$$

Pareto optimality is defined when $\boldsymbol{\alpha} \in \mathbb{R}^p_+$ such that $\alpha_i > 0 \forall i = 1, \ldots, p$. This restricts the Pareto frontier to lie in the strictly positive orthant in $\mathbb{R}^p$. However, Yu introduced the concept of Λ-extreme points to the multi-objective optimization problems as a more general definition of Pareto optimality on the basis of a cone Λ [106, 107]. This allows for $\boldsymbol{\alpha} \in \mathbb{R}^p \backslash \{0\}$ and a concept of Pareto optimality over the entire objective space $\mathbb{R}^p$.

A distinction needs to be made between the Pareto frontier and the feasible objective space, $Y = \mathbf{J}[D]$. The Pareto frontier is a subset of the objecitve space that lies in the strictly positive orthant in $\mathbb{R}^p$ and contains non-dominated points. Λ-extreme points and generalized hyperplane methods allow the concept of Pareto dominance and optimality to extend to more regions of the objective space.

The first-order necessary conditions of optimality presented in both Eq. (4.11) and Eq. (4.17) share the same form. For the reachability optimal control problem, the initial states $\mathbf{x}_0 = \mathbf{x}(t_0)$ correspond to the design variables $\mathbf{d}$ as these are the free variables that need to be determined to optimize the problem at hand. The initial states must also satisfy the constraints given, making $\mathbf{g}(\mathbf{x}_0), \boldsymbol{\lambda}$ equivalent to $\mathbf{h}(\mathbf{d}), \boldsymbol{\mu}$, respectively]. For minimum-time reachability optimal control problems, the objective functions $\mathbf{J}$ are the final states $\mathbf{x}_f = \mathbf{x}(t_f) = \phi_x(t_f)$. Finally, these objectives are scalarized by an weighted sum as in Eq. (4.10) and Eq. (4.15) making $\hat{\mathbf{d}}_s$ equivalent to $\boldsymbol{\alpha}$.

Rewriting the multi-objective optimization necessary condition in Eq. (4.14a) with the discussed equivalencies results in

$$\frac{\partial \mathbf{x}_f}{\partial \mathbf{x}_0}^T \hat{\mathbf{d}}_s + \frac{\partial \mathbf{g}}{\partial \mathbf{x}_0}^T \boldsymbol{\lambda} = 0 \tag{4.18}$$

From the optimal control theory formulation in §4.1, Eq. (4.4e) is used to rewrite the multi-objective necessary condition of optimality in Eq. (4.18) as

$$\frac{\partial \mathbf{x}_f}{\partial \mathbf{x}_0}^T \hat{\mathbf{d}}_s = \frac{\partial V}{\partial \mathbf{x}_0} = \mathbf{p}_0 \tag{4.19}$$

From optimal control theory, a well-known result comes from the Hamilton-Jacobi-Bellman partial differential equation (HJB PDE) [72, 105]

$$\frac{\partial V}{\partial t} + \max_{\mathbf{u}\in U}\left[\mathcal{L}(\mathbf{x},\mathbf{u},t) + \frac{\partial V}{\partial \mathbf{x}}\,\mathbf{f}(\mathbf{x},\mathbf{u},t)\right] = \frac{\partial V}{\partial t} + \max_{\mathbf{u}\in U}\big[\mathcal{H}(\mathbf{x},\frac{\partial V}{\partial \mathbf{x}},\mathbf{u},t)\big] = 0 \tag{4.20}$$

where $V(\mathbf{x},t)$ denotes the value function. The value function describes the optimal cost-to-go from the current state and time $(\mathbf{x},t)$ where the cost is defined by the optimal control problem in Eq. (4.1). The HJB PDE in Eq. (4.20) is a necessary condition of optimality along all optimal trajectories. Furthermore, it can be shown that the costate is equivalent to the gradient of this value function with respect to the state such that $\mathbf{p} = \partial V/\partial \mathbf{x}$ [105]. Consequently, for the minimum-time reachability problem, Eq. (4.19) corresponds to the necessary condition of optimality for the reachability optimal control problem. This supports that the multi-objective and reachability optimal control formulations have equivalent necessary conditions of optimality. With this, they are now combined into a single formualtion.

4.3 Joint Reachability and Multi-objective Optimization Formulation - HJB PDE

The following section will introduce a joint formulation for simultaneously dealing with multi-objective optimization and optimal control problems of the form Eq. (4.1) and Eq. (4.13). Collect all decision variables within an augmented state

$$\mathbf{X} = [\mathbf{x}^T\ \mathbf{J}^T\ \mathbf{d}^T]^T \tag{4.21}$$

with dynamical states $\mathbf{x} \in \mathbb{R}^n$, objective functions $\mathbf{J} \in \mathbb{R}^p$, design parameters $\mathbf{d} \in \mathbb{R}^d$, and $n + p + d = k$ ($\mathbf{X} \in \mathbb{R}^k$).

To generalize the problem and avoid explicit time-dependent formulations, the independent parameter q is introduced. The generalized independent parameter (GIP) q is defined as

$$q = Q(\mathbf{X}(t), \mathbf{u}(t), t) = \int_{t_0}^{t} l(\mathbf{X}(\tau), \mathbf{u}(\tau), \tau) d\tau \tag{4.22}$$

where it is assumed the mapping Q between $q \in [q_0, q_f]$ to $t \in [t_0, t_f]$ is both one-to-one and onto for the simplicity. Note that unique mapping between q and t is not required for the formulation as shown by Holzinger et al. [70]. In this paper, it was shown that the mapping Q is required to be either fully invertible or only left invertible [70]. The inclusion of this generalized independent parameter q is a generalization of the the equivalency between the Bolza and Mayer problems of optimal control theory [72, 108]. Moreover, this independent parameter allows for problem-specific independent variables such as cost or fuel usage. If time-dependent dynamics are desired, choose $l(\cdot) = 1$ such that $q = t$.

The augmented state dynamics with respect to the GIP q can be written as

$$\mathbf{X}' = \frac{d\mathbf{X}}{dq} = \frac{\frac{d\mathbf{X}}{dt}(\mathbf{X}(q), \mathbf{u}(q), q)}{\frac{dq}{dt}(\mathbf{X}(q), \mathbf{u}(q), q)} = \tilde{\mathcal{F}}(\mathbf{X}, \mathbf{u}, q) \tag{4.23}$$

Here the $\tilde{\cdot}$ notation indicates that a function uses q as the independent parameter either through a natural parametrization or from a transformation such as Eq. (4.23). The combined dynamics of $\mathbf{X}$ with respect to q are

$$\frac{d\mathbf{X}}{dq} = \begin{bmatrix} \frac{d\mathbf{x}}{dq} \\ \frac{d\mathbf{J}}{dq} \\ \frac{d\mathbf{d}}{dq} \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{f}}(\mathbf{X}, \mathbf{u}, q) \\ \tilde{\mathcal{L}}(\mathbf{X}, \mathbf{u}, q) \\ \tilde{\mathbf{S}}(\mathbf{X}, q) \end{bmatrix} \tag{4.24}$$

Suppose the surface of extremal solutions is known such that $g(\mathbf{X}(q = q_0)) = 0 =$

$V(\mathbf{X}_0, q_0)$. Through an application analogous to minimum-time reachability theory [5, 1], the resulting set of extremal solutions to the unified formulation may now be written in the form of a reachability problem, which is the solution to the Generalized Independent Parameter (GIP) HJB PDE expressed in Eq. (4.25) [70].

$$\frac{\partial V}{\partial q} + \max_{\mathbf{u} \in U} \left[\frac{\partial V}{\partial \mathbf{X}} \frac{d\mathbf{X}}{dq} \right] = 0 \tag{4.25}$$

The zero-level sets $V(\mathbf{X}, q) = 0$ implicitly define the extremal surface over $\mathbf{X}$ [5]. Expanding Eq. (4.25) results in the following equation denoted as the joint GIP HJB PDE:

$$\frac{\partial V}{\partial q} + \max_{\mathbf{u} \in U} \left[\frac{\partial V}{\partial \mathbf{x}} \, \tilde{\mathbf{f}}(\mathbf{X}, \mathbf{u}, q) + \frac{\partial V}{\partial \mathbf{J}} \, \tilde{\mathcal{L}}(\mathbf{X}, \mathbf{u}, q) + \frac{\partial V}{\partial \mathbf{d}} \, \tilde{\mathbf{S}}(\mathbf{X}, q) \right] = 0 \tag{4.26}$$

4.3.1 Reduction of Joint Formulation to Minimum-Time Reachability

The joint GIP HJB PDE reduces to the GIP-based optimal control problem when there are no design variables and a single objective function defined as in Eq. (4.1). In this case, the joint GIP HJB PDE reduces to

$$\frac{\partial V}{\partial q} + \max_{\mathbf{u} \in U} \left[\tilde{\mathcal{L}}(\mathbf{x}, \mathbf{u}, q) + \frac{\partial V}{\partial \mathbf{x}} \, \tilde{\mathbf{f}}(\mathbf{x}, \mathbf{u}, q) \right] = \frac{\partial V}{\partial q} + \max_{\mathbf{u} \in U} \left[\tilde{\mathcal{H}}(\mathbf{x}, \frac{\partial V}{\partial \mathbf{x}}, \mathbf{u}, q) \right] = 0 \tag{4.27}$$

which is the classical HJB PDE derived from optimal control theory expressed with respect to GIP [72, 109, 105].

Similar to traditional optimal control reachability, the minimum-time reachability problem can be acquired from the classical GIP HJB PDE in Eq. (4.27). When the GIP mapping function integrand is the objective function Lagrangian ($l(\mathbf{x}(t), \mathbf{u}(t), t) = \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t)$) and $Q(\mathbf{x}(t), \mathbf{u}(t), t)$ is monotonic [71]. This case is discussed by Holzinger et al. as the definition of the Generalized Metric Range Sets where the independent parameter q is equivalent to the objective function itself [71, 70]. Considering this case without the inclusion of

design parameters, the joint GIP HJB PDE in Eq. (4.26) reduces to

$$\frac{\partial V}{\partial q} + 1 + \max_{\mathbf{u} \in U} \left[\frac{\partial V}{\partial \mathbf{x}} \, \tilde{\mathbf{f}}(\mathbf{x}, \mathbf{u}, q) \right] = 0 \tag{4.28}$$

As the 1 does not affect the optimal control policy, state dynamics, or costate dynamics, Eq. (4.28) is equivalent to

$$\frac{\partial V}{\partial q} + \max_{\mathbf{u} \in U} \left[\frac{\partial V}{\partial \mathbf{x}} \, \tilde{\mathbf{f}}(\mathbf{x}, \mathbf{u}, q) \right] = 0 \tag{4.29}$$

which is the GIP minimum-time reachability problem. Given an initial extremal reachable set such that $g(\mathbf{x}(q = q_0)) = 0 = V(\mathbf{x}_0, q_0)$, the zero-level set of the $V(\mathbf{x}, q)$ can be computed from Eq. (4.29) and corresponds to the boundary of the GIP reachable set. Traditional minimum-time reachability is performed in the case when $q = t$.

4.3.2 Reduction of Joint Formulation to Multi-objective Optimization

In a similar manner, consider the traditional multi-objective optimization case where there are no state dynamics and but there are objective functions and design variables. Homotopy, in general, involves the deformation of one continuous function to another [110]. This field is commonly used to solve difficult problems by solving a simpler ones and deforming the simple solutions to the difficult solutions [111]. To allow for GIP-based dynamics for the objective function, one may construct objective functions as the following linear homotopy map [20, 112]

$$\mathbf{J}(\mathbf{d}, q) = (1 - q)\mathbf{J}_0(\mathbf{d}) + q\mathbf{J}_f(\mathbf{d}) \tag{4.30}$$

where $q \in [0, 1]$, $\mathbf{J}_0(\mathbf{d})$ denotes the initial objective function, and $\mathbf{J}_f(\mathbf{d})$ denotes the terminal objective function. This homotopy map is chosen such that when $q = 0$, $\mathbf{J}(\mathbf{d}) = \mathbf{J}_0(\mathbf{d})$ and when $q = 1$, $\mathbf{J}(\mathbf{d}) = \mathbf{J}_f(\mathbf{d})$. The GIP serves the role of the homotopy/continuation parameter in this problem formulation. To specify an initial condition on the joint GIP

HJB PDE, one should define $\mathbf{J}_0(\mathbf{d})$ such that $V(\mathbf{J}_0, \mathbf{d}_0, 0) = 0$ is well-defined and known. The terminal objective function $\mathbf{J}_f(\mathbf{d})$ should be set to the original, given set of objective functions one wants to optimize. Using this linear homotopy map, the objective function GIP dynamics are

$$\frac{d\mathbf{J}}{dq}(\mathbf{d}) = \mathbf{J}_f(\mathbf{d}) - \mathbf{J}_0(\mathbf{d}) \tag{4.31}$$

where the resulting, multi-objective optimization form of the joint GIP HJB PDE is given as

$$\frac{\partial V}{\partial q} + \frac{\partial V}{\partial \mathbf{J}} \frac{d\mathbf{J}}{dq}(\mathbf{d}) + \frac{\partial V}{\partial \mathbf{d}} \tilde{\mathbf{S}}(\mathbf{J}, \mathbf{d}, q) = 0 \tag{4.32}$$

Eq. (4.32) describes a GIP-based minimum-time reachability problem. However, in this case, the reachable set is computed over $\mathbf{X} = [\mathbf{J}^T \mathbf{d}^T]^T \in \mathbb{R}^{p+d}$ representing the objective space and feasible design space. Just as in the previous minimum-time reachability discussion, the zero-level sets of $V(\mathbf{x}, q)$ describe the boundary of the reachable set of $(\mathbf{J}, \mathbf{d})$. As this set contains all possible values of objective functions and design parameters, the projection onto the $\mathbf{J}$-subspace in $\mathbb{R}^p$ results in the feasible objective function set.

The concept of Pareto-optimality and more generally Λ-extremity, directly depends on the feasible objective function set. This is because the Pareto-optimal set, and by extension the Λ-extreme set, is always on the boundary of the feasible objective function set [113]. Through post-processing of the boundary of the feasible objective function set, these Pareto-optimal sets can be computed. If the feasible objective function set is convex, then the boundary of this set is the Λ-extreme set and the Pareto-optimal set is an orthant section of this set.

Feasibility constraints on the design parameters and objective functions should be considered when formulating the multi-objective optimization GIP HJB PDE in Eq. (4.32). One method to account for constraints is by carefully formulating $V(\mathbf{J}_0, \mathbf{d}_0, 0) = 0$ as well as the GIP dynamics for the design variables. Another technique is to add constraints to the

boundary conditions and/or the GIP Hamiltonian. This approach is common within the field of Hamilton-Jacobi (HJ) reachability for problems related to reachable tube computation, collision avoidance, target sets, system safety assurance, obstacle sets, and reach-avoid sets [114, 28, 115, 73, 1, 29].

With the full, joint GIP HJB PDE given in Eq. (4.26), joint analyses of multi-objective optimization, optimal control, and reachability problems are performed simulataneously. Example applications come from the fields of trajectory optimization, multi-objective optimal control, generalized range set over multiple range metrics, and path-planning.

The cost of computing this type of analysis comes in the form of the larger augmented state dimension. Typically, HJ-based methods do not scale well in terms of state dimension. However, advancements in HJ-reachability such as projective methods and system decomposition techniques have helped alleviate the curse of dimensionality by reducing the effective state dimension in the computations [10, 114, 60, 116]. However, for nonlinear systems the cost of these methods are still exponential in the effective state space dimension and time.

4.4 Solution Methods for Joint Reachability and Multi-objective Optimization Formulation

Through the first-order necessary conditions of optimality and through the joint formulation, this chapter has identified analytic connections between reachability and optimal control with multi-objective optimization. Both of these fields have a vast amount of research in the development of numerical solution methods to solve these problems. If a problem from the multi-objective optimization field may be cast into a reachability problem and vice-versa, there is a potential to have cross-pollination of the numerical techniques used to solve these problems. This section will give examples of how solution techniques and algorithms of one field may be applied to problems of the other.

4.4.1 Solving Reachability Problems with Multi-objective Optimization

Consider the initial value problem for a dynamic system of the form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t), \mathbf{x}(t_0) = \mathbf{x}_0, \mathbf{u} \in U \tag{4.33}$$

If the dynamic system has state dynamics that are Lipschitz continuous in the state and continuous in time, then the Picard-Lindelöf theorem states the initial value problem has a unique solution [97]. Under these conditions, one may explicitly write

$$\mathbf{x}_f = \mathbf{x}(t_f) = \int_{t_0}^{t_f} \mathbf{f}(\mathbf{x}(\cdot), \mathbf{u}(\cdot), \tau) d\tau + \mathbf{x}_0 = \phi_x(t_f; \mathbf{x}_0) \tag{4.34}$$

where ϕ_x denotes the trajectory flow function for the dynamic states. In general, the solution $\mathbf{x}_f$ does not have an analytical form and must be numerically integrated using Eq. (4.34). If the control policy $\mathbf{u}(\cdot)$ is known through controller design or optimal control, one may uniquely determine the final state $\mathbf{x}_f$ given the initial state $\mathbf{x}_0$ and a propagation time t_f.

For minimum-time reachability problems, the optimal control policy can be determined using Eq. (4.4c). Even so, the optimal control policy does not always have an analytic solution. In cases like this, one must solved a constrained single-objective optimization problem specified in Eq. (4.4c). However, for many dynamic systems including control affine systems, analytic results for this optimal control policy can be computed [63, 117].

With this, minimum-time reachability problems can be put in the multi-objective optimization framework described in §4.2 by defining the following objectives and design variables

$$\begin{aligned} \mathbf{J} &= \mathbf{x}_f = \phi_x(t_f; \mathbf{x}_0, \boldsymbol{\lambda}) \\ \mathbf{d} &= [\mathbf{x}_0^T, \boldsymbol{\lambda}^T]^T \\ \mathbf{h} &= \mathbf{g} \end{aligned} \tag{4.35}$$

where $\phi_x(t_f; \mathbf{x}_0, \boldsymbol{\lambda})$ is typically computed through numerical integration of the initial value problem. Subspace reachable sets may be computed by selectively reducing the dimension of the objective function. This can be performed by defining the objective function to be

$$\mathbf{J} = S\mathbf{x}_f \tag{4.36}$$

where $S \in \mathbb{R}^{s \times n}$, $s \leq n$, and the rows of S correspond to rows of the identity matrix $\mathbb{I}_{n \times n}$ that belong to the subspace of interest.

The reachability multi-objective optimization problem is then formally stated as

$$\begin{aligned} \max_{\mathbf{d}} \quad & \mathbf{J}(\mathbf{d}) \\ s.t. \quad & \mathbf{h}(\mathbf{d}) = \mathbf{0} \end{aligned} \tag{4.37}$$

Once in this form, minimum-time reachability problems may be analyzed by using any number of available multi-objective optimization techniques. Common solution techniques include evolutionary algorithms, scalarization methods, normal boundary intersection, ϵ-constraint methods, and homotopy-based algorithms [48, 49, 50, 44, 51, 52, 53, 54, 55, 56].

Many of these algorithms are performed over orthants of the objective space. To create a reachable set, one must repeat the multi-objective optimization algorithm over all of the orthants of interest. For example, in a two-dimensional reachability analysis, one must perform a multi-objective optimization over the four quadrants of the state/objective space.

Another thing to consider is that many of these methods search for sets of non-dominated

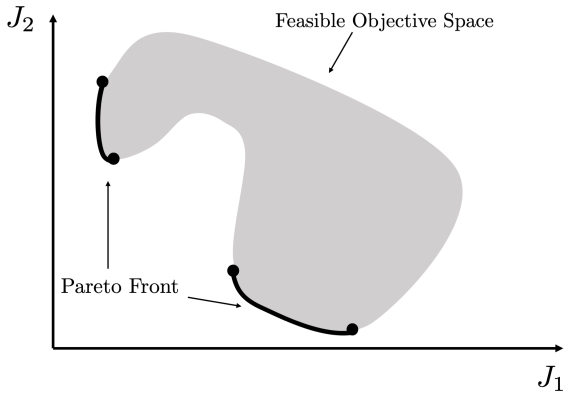


Figure 4.1: Example Pareto front with non-convex region causing gap in surface

points within the objective function space based on Pareto dominance order. For convex Pareto fronts, this is equivalent to the boundary of the feasible objective space. However, depending on the degree of non-convexity and orientation of the feasible objective set, there may be gaps in the Pareto front, as demonstrated in Figure 4.1. While the complete description of the boundary of the reachable set is important in a reachability analysis, Pareto dominance order restricts the extremal set when the curvature of the non-convex regions is too large [118]. One technique to avoid this is by implementing different dominance rules such as Λ-extremity or generalized hyperplanes [106, 107]. Another method of dealing with this problem is in the numerical implementation, as many of the Pareto-based algorithms compute these non-Pareto boundary points but subsequently disregard them.

4.4.2 Solving Multi-objective Optimization Problems with Reachability

To compute multi-objective optimization problems in a reachability framework, one can first express the objective functions as a linear homotopy map such that

$$\begin{aligned} \mathbf{J}(\mathbf{d}, q) &= (1-q)\mathbf{J}_0(\mathbf{d}) + q\mathbf{J}_f(\mathbf{d}) \\ \frac{d\mathbf{J}}{dq}(\mathbf{d}) &= \mathbf{J}_f(\mathbf{d}) - \mathbf{J}_0(\mathbf{d}) \end{aligned} \tag{4.38}$$

where $q \in [0,1]$, $\mathbf{J}_0(\mathbf{d})$ is decided by the user, and $\mathbf{J}_f(\mathbf{d}) = \mathbf{J}(\mathbf{d})$ represents the known/given vector of objective functions. The design variable dynamics $\tilde{\mathbf{S}}(\mathbf{J}, \mathbf{d}, q)$ is also decided by the user. These dynamics must be chosen such that the design variables remain within the feasible design space for all values of q. This may lead to design variable dynamics of the form of artificial potential functions or barrier certificates [119, 120].

Once these parameters are all defined, the multi-objective optimization problem may be solved in a number of methods throughout reachability theory. For example, one may use the joint GIP HJB PDE reiterated here

$$\begin{aligned} \frac{\partial V}{\partial q} + \frac{\partial V}{\partial \mathbf{J}} \frac{d\mathbf{J}}{dq}(\mathbf{d}) + \frac{\partial V}{\partial \mathbf{d}} \tilde{\mathbf{S}}(\mathbf{J}, \mathbf{d}, q) = 0 \\ V(\mathbf{J}_0, \mathbf{d}_0, q_0) = 0 \end{aligned} \tag{4.39}$$

Within the realm of HJ-reachability analyses, level set methods are common solution methods [7]. However, there are many of analytic and numerical techniques for solving first order PDEs as the one in Eq. (4.39).

Other reachability analysis techniques require variable dynamics, limits on control inputs, and variable initial condition sets. With these known, there are a large number of reachability tools and algorithms available that would solve for the extremal surface as a function of q. The reachability problem can be stated as

$$\begin{aligned} \max_{\mathbf{d}} \quad & \begin{bmatrix} \mathbf{0}_{p\times p} & \mathbb{I}_{p\times d} \\ \mathbf{0}_{d\times p} & \mathbf{0}_{d\times d} \end{bmatrix} \mathbf{X} \\ s.t. \quad & \mathbf{g}(\mathbf{X}) = \mathbf{0} \\ & \mathbf{X} = [\mathbf{J}^T, \mathbf{d}^T]^T \end{aligned} \tag{4.40}$$

where $\mathbb{I}$ denotes the identity matrix. In general, the main tradeoff with reachability algorithms revolves around computational speed and solution accuracy. However, there have

been some recent developments in computational reachability that allow for efficient and accurate computation of reachable sets [11, 12, 13, 121, 14, 122].

The resulting extremal set will depend on both the algorithm used and the internal assumptions of the algorithm. If the true extremal set is non-convex, the algorithms may return the true boundary of the extremal set. Similar in the discussion of §4.4.1, the boundary of the extremal set and the Pareto front are not guaranteed to be the same. As a result, if the Pareto front is desired, post processing of the extremal sets may be performed to recover the desired results.

4.5 Joint Reachability and Multi-objective Optimization Formulation - Continuation Solution Technique

This next section presents an alternative solution technique on the joint system given by Eq. (4.21) and Eq. (4.24) based on the principles of numerical continuation and homotopy.

The purpose of continuation methods is to solve a nonlinear system of equations by solving a simpler one and deforming the simple solution to the more complex solution. This deformation is generally performed using a scalar continuation parameter which parametrizes the root-finding problem. Numerical continuation methods have seen many applications in the study of dynamical systems particularly in chaotic system analysis and parametric bifurcation as well as in modern application in the search for quasi-periodic invariant tori in the circular restricted three body problem [20, 22, 123].

HJ-reachability problems are typically solved using an Eulerian-type approach because they require gridding the state space resulting in time and memory requirements that scale exponentially with the state dimension. On the other hand, Lagrangian methods do not depend on the gridding of state space, allowing for computational feasible analyses of high-dimensional systems.

Of the Lagrangian-type approaches, Brew et al. have presented continuation-based reachability methodologies that compute samples of the reachable set boundary convex hull [63, 64]. The algorithm is titled Sampled Continuation Reachability (SCoRe) method. This sample-based method allows for the computation of subspace reachable sets, effectively reducing the state dimension cost to the dimension of the subspace of interest. As this approach uses the first-order necessary conditions of optimality for the reachability problem, guarantees of accuracy and optimality can be made. Furthermore, by construction, the presented reachability algorithm is parallelizable.

Given the definitions of the joint state and its dynamics shown in Eq. (4.21) and Eq. (4.24), the joint continuation reachability optimal control problem is defined as

$$\begin{aligned} &\sup_{\mathbf{u}\in U} \hat{\mathbf{d}}(\boldsymbol{\theta})^T \mathbf{X}_f \\ s.t. \quad &\mathbf{X}' = \frac{d\mathbf{X}}{dq} = \tilde{\mathcal{F}}(\mathbf{X}, \mathbf{u}, q) \\ &\mathbf{g}(\mathbf{X}_0) = \mathbf{0} \end{aligned} \tag{4.41}$$

where $\mathbf{g} \in \mathbb{R}^v$, $\hat{\mathbf{d}} = \mathbf{h}(\boldsymbol{\theta}) \in \mathbb{S}^{n+p+d}$ denotes a unit search direction, $\boldsymbol{\theta} \in \mathbb{R}^{n+p+d-1}$ is a vector of angular coordinates, $\mathbb{S}$ denotes the unit hypersphere, and $\mathbf{h}$ is the mapping from the angular coordinates to a unit vector.

Using the results discussed in Section 4.1, the first-order necessary conditions of optimality

for Eq. (4.41) are

$$\mathbf{X}' = \frac{\partial \tilde{\mathcal{H}}^*}{\partial \mathbf{P}}^T \tag{4.42a}$$

$$\mathbf{P}' = -\frac{\partial \tilde{\mathcal{H}}^*}{\partial \mathbf{X}}^T \tag{4.42b}$$

$$\mathbf{u} = \underset{\mathbf{u} \in U}{\text{argmax}} \{\tilde{\mathcal{H}}\} \tag{4.42c}$$

$$\mathbf{P}_f = \hat{\mathbf{d}}(\boldsymbol{\theta}) \tag{4.42d}$$

$$\mathbf{P}_0 = -\frac{\partial \mathbf{g}}{\partial \mathbf{X}_0}^T \boldsymbol{\lambda} \tag{4.42e}$$

$$\mathbf{g}(\mathbf{X}_0) = \mathbf{0} \tag{4.42f}$$

where $\mathbf{P} \in \mathbb{R}^{n+p+d}$ denotes the joint costate variable and $\boldsymbol{\lambda} \in \mathbb{R}^v$ denotes the Lagrange multiplier associated with the initial condition constraint. Assuming joint state dynamics are Lipschitz continuous, the following system of equations represents the necessary conditions

$$\mathbf{F}(\mathbf{Z}; \hat{\mathbf{d}}(\boldsymbol{\theta})) = \mathbf{F}(\mathbf{X}_0, \boldsymbol{\lambda}; \hat{\mathbf{d}}(\boldsymbol{\theta})) = \begin{bmatrix} \hat{\mathbf{d}}(\boldsymbol{\theta}) - \phi_P(t_f; \mathbf{X}_0, \mathbf{P}_0(\mathbf{X}_0, \boldsymbol{\lambda})) \\ \mathbf{g}(\mathbf{X}_0) \end{bmatrix} = \mathbf{0} \tag{4.43a}$$

$$\mathbf{Z} = \begin{bmatrix} \mathbf{X}_0 \\ \boldsymbol{\lambda} \end{bmatrix} \tag{4.43b}$$

where $\mathbf{Z} \in \mathbb{R}^{n+v}$ denotes the joint reachability optimal control problem solution vector.

For every sampled search unit vector $\hat{\mathbf{d}}(\boldsymbol{\theta})$, a single objective optimal control problem is created. The solution to this optimal control problem results in a sample of the reachable set boundary using the methods discussed in [63, 64]. For a given sample, when a component of $\hat{\mathbf{d}}$ is zero, the corresponding component of $\mathbf{X}_f$ does not contribute to the reachability objective function. This is entirely analogous to the scalarized objective function in Eq. (4.15) used in the KKT conditions. Similarly, the generated reachable set boundary samples do

not sample non-convex regions of the reachable set.

One of the advantages of continuation-based methods is that the continuation parameter itself does not have to be time- or GIP-based. For example, Brew et al. proved that continuation can be performed on the $\boldsymbol{\theta}$ or $\hat{\mathbf{d}}$ variables. This is equivalent to locally exploring the reachable set. A similar concept is thoroughly discussed by Hillermeier in his use of homotopy methods for computing Pareto-frontiers in multi-objective optimization problems [55]. This allows for the quantification of efficient tradeoffs on the extremum surface as well as the sensitivity of these tradeoffs.

4.6 Results

4.6.1 Hillermeier academic example

In Hillermeier §7.1, an academic example of multi-objective optimization is given to demonstrate the use of homotopy/continuation methods to compute a Pareto-frontier [55]. The optimization problem of two objectives as given as

$$\mathbf{J} = \begin{cases} \mathbb{R}^2 \to \mathbb{R}^2 \\ \mathbf{d} \to \mathbf{J}(\mathbf{d}) = [J_1(\mathbf{d}), J_2(\mathbf{d})]^T \end{cases} \tag{4.44}$$

where

$$\begin{aligned} J_1(\mathbf{d}) &= \cos(a(\mathbf{d})) \cdot b(\mathbf{d}) \\ J_2(\mathbf{d}) &= \sin(a(\mathbf{d})) \cdot b(\mathbf{d}) \\ a(\mathbf{d}) &= \frac{2\pi}{360}[a_c + a_1 \sin(2\pi d_1) + a_2 \sin(2\pi d_2)] \\ b(\mathbf{d}) &= 1 + c\cos(2\pi d_1) \end{aligned} \tag{4.45}$$

The chosen values for this demonstration are $a_c = 45, a_1 = 40, a_2 = 25$, and $c = 0.5$. As this problem is periodic with period 1 in terms of d_1 and d_2, their feasible range is

unlimited.

For this problem, the linear homotopy map from Eq. (4.30) is created for the objective function GIP dynamics. The initial and terminal homotopy objective functions are defined as

$$\begin{aligned} J_{1,0}(\mathbf{d}) &= 0 \\ J_{2,0}(\mathbf{d}) &= 0 \\ J_{1,f}(\mathbf{d}) &= \cos(a(\mathbf{d})) \cdot b(\mathbf{d}) \\ J_{2,f}(\mathbf{d}) &= \sin(a(\mathbf{d})) \cdot b(\mathbf{d}) \end{aligned} \tag{4.46}$$

The chosen GIP dynamics for the design variables are $\tilde{\mathbf{S}}(\mathbf{J}, \mathbf{d}, q) = 0$. This choice is so the feasible set of design variables is determined by the initial condition set.

To approximate the conditions specified above with a convex, smooth, and differentiable set, the initial condition constraint is chosen as

$$g(\mathbf{X}_0) = \left(\frac{J_{1,0}}{\epsilon}\right)^2 + \left(\frac{J_{2,0}}{\epsilon}\right)^2 + \left(\frac{d_{1,0}}{0.5}\right)^2 + \left(\frac{d_{2,0}}{0.5}\right)^2 = 0 \tag{4.47}$$

where $\epsilon << 1$. With ϵ = 1e-3, the initial values of $\mathbf{J}$ when $q = 0$ are small. While 0.5 was chosen as the maximum initial distance from the origin for the design variables, any value greater than 0.5 results in the same results due to the periodicity of the objective functions.

To demonstrate the topics discussed in this chapter, the boundary of the feasible objective function set will be sampled using the SCoRe reachability algorithm described in Section 4.5. The feasible objective function space, $\mathbf{J}(\mathbf{d} \in D)$ was generated by making an fine grid of values from $d_1, d_2 \in [0, 1]$ and evaluating the objective functions. With 60 unit vectors evenly spaced in the $\mathbf{J}$ subspace, the reachable set samples were computed. The results are displayed in Figure 4.2. This computation was completed in 10.5 seconds on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor with 8 GB 2133 MHz RAM.

The samples computed by the SCoRe algorithm provide a convex representation of the

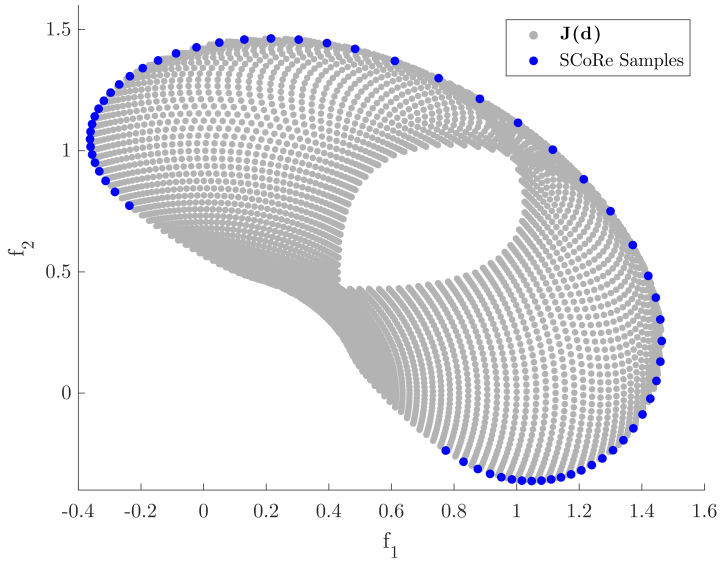


Figure 4.2: Example 7.1 from Hillermeier generated from sampling design space and from the SCoRe algorithm

feasible objective function space. Because of the support function based construction of the SCoRe algorithm, the non-convex region close to the origin of the objective function space is not sufficiently sampled. This result is still a proof-of-concept for computing extremal objective function sets using a reachability algorithm and homotopy methods. Further validation may be performed by computing the reachable set using an HJ-reachability technique.

4.6.2 Cislunar Space Problem Trajectory Optimization

The joint extremal surfaces generated using the presented methodology provide the set of extremal solutions for the given objective functions. As this formulation includes dynamic states driven through optimal control, solutions to multi-objective optimal control problems are found. Moreover, Pareto-efficient tradeoffs between the objective functions are found. For instance, sample objetive functions for a multi-objective optimal control problem may be a minimum L_2 (or L_1) control effort and minimum time. Using the continuation-based methodologies discussed, the extremal solution surfaces describing tradeoffs between min-

imum control effort and minimum time state trajectories are found. Furthermore, the corresponding optimal control policies for these Pareto-optimal solutions are computed as well.

This is demonstrated on the problem of a spacecraft in orbital motion affected by both the Earth and the Moon. The non-dimensional nonlinear equations of motion for a body in the plane of the Earth-moon system are described as

$$\begin{aligned} \overset{\circ\circ}{x} &= 2\overset{\circ}{y} + x - (1-\mu)\frac{x-x_1}{\rho_1^3} - \mu\frac{x-x_2}{\rho_2^3} + \bar{u}_x \\ \overset{\circ\circ}{y} &= -2\overset{\circ}{x} + \left(1 - \frac{1-\mu}{\rho_1^3} - \frac{\mu}{\rho_2^3}\right) y + \bar{u}_y \end{aligned} \tag{4.48}$$

where (x, y) are normalized position components relative to the Earth-Moon rotating frame, μ is the non-dimensional mass ratio, $x_1 = -\mu$, $x_2 = 1 - \mu$, $\rho_i = \sqrt{(x - x_i)^2 + y^2}$ is the non-dimensional relative distance, $(\bar{u}_x, \bar{u}_y)$ denote non-dimensional acceleration control input,and $\overset{\circ}{(\cdot)}$ denotes derivatives with respect to the non-dimensional time variable, τ [80]. For detailed derivations of these dynamics, please refer to [80].

This example will analyze a multi-objective optimal control problem with minimum time and minimum fuel objectives. In terms of the joint formulation, this results in the following augmented state

$$\mathbf{X} = [x \;\; y \;\; \overset{\circ}{x} \;\; \overset{\circ}{y} \;\; J_1]^T \tag{4.49}$$

where the fuel usage cost is represented by J_1 as

$$J_1 = -\int_0^{\tau_f} \|\bar{\mathbf{u}}\|_1 d\tau = -\int_0^{\tau_f} |\bar{u}_x| + |\bar{u}_y| d\tau \tag{4.50}$$

The true acceleration with respect to time as opposed to the non-dimensional time is computed by rescaling the non-dimensional quantities

$$\mathbf{u} = \omega^2 r_{12} \bar{\mathbf{u}} \tag{4.51}$$

where $r_{12} = 384402$ km and $\omega = 13.19$ degrees per sidereal day for the Earth-Moon system. This allows the total ΔV consumed in a given trajectory to be expressed as

$$\Delta V = \int_0^{t_f} \|\mathbf{u}\|_1 dt = \omega r_{12} \int_0^{\tau_f} |\bar{u}_x| + |\bar{u}_y| d\tau = -\omega r_{12} J_1 \tag{4.52}$$

Lagrange points are points near two large astronomical bodies where a smaller object will maintain its position relative to the larger bodies [80]. In terms of the dynamic system given in Eq. (4.48), the Lagrange points correspond to equilibrium points. Earth-Moon Lagrange points are desirable locations for spacecraft because of the favorable maneuverability between the Earth and the Moon, the capability to collect lunar and space weather observations from a fixed vantage point, and the possible location of refueling and servicing space stations [124, 125, 126].

In the following scenario, a low-thrust spacecraft is placed near the first Lagrange point of the Earth-Moon system, denoted by L1. The coordinates for this stationary point are $x_{L1} = 0.836915$ and $y_{L1} = 0$. The spacecraft thrust magnitude limits are modeled after the Lunar IceCube 6U CubeSat with maximum thrust of 0.8 mN for a spacecraft of a mass of 14 kg [127]. However, in this case a two-thruster configuration is invetigated where each thruster is independently controlled and axis-aligned with the Earth-moon rotating frame.

The initial condition set for the augmented state in Eq. (4.49) is approximated using an ellipsoid

$$g(\mathbf{X}_0) = \left(\frac{x - x_{L1}}{\epsilon_1}\right)^2 + \left(\frac{y}{\epsilon_1}\right)^2 + \left(\frac{\mathring{x}}{\epsilon_2}\right)^2 + \left(\frac{\mathring{y}}{\epsilon_2}\right)^2 + \left(\frac{J_1}{\epsilon_2}\right)^2 = 0 \tag{4.53}$$

where $\epsilon_1 = 1e-3$ and $\epsilon_2 = 1e-4$. This provides a smooth, differentiable initial condition set that represents a spacecraft placed near L1 with some insertion errors with independent standard deviations given by ϵ_1 and ϵ_2.

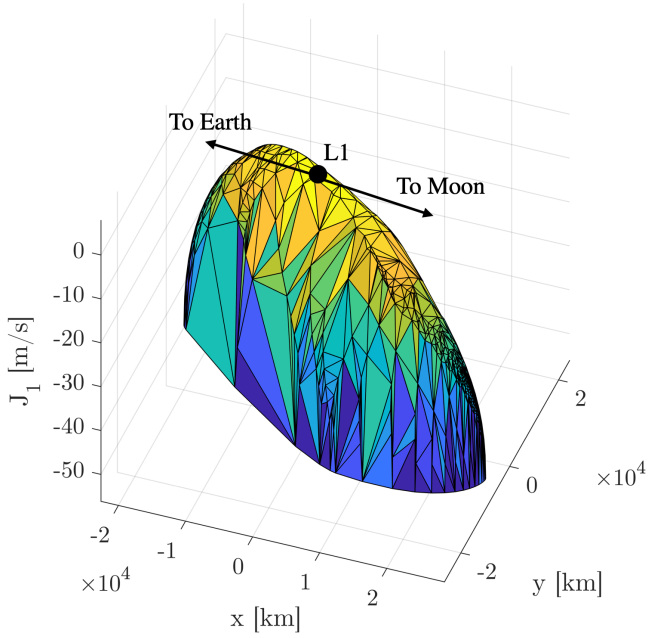


Figure 4.3: Minimum fuel and minimum-time reachability tradeoffs

The problem at hand is to perform a 5-day maneuverability analysis for the spacecraft in this scenario. This involves computing the 5-day reachable set of the spacecraft as a function of the ΔV used. Additionally, one would like to determine efficient tradeoffs between the reachable states and the ΔV cost it takes to achieve that state. To answer these questions simulataneously, a subspace reachability analysis is performed on the x, y, J_1 space. This reachability analysis is performed using the continuation methods described in §4.5 with 600 particles. Because the J_1 cost monotonically decreases with use of control input, with search directions $\hat{\mathbf{d}}_s$ are sampled from the semisphere corresponding to $J_1 < 0$. This computation was completed in about 1200 seconds on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor with 8 GB 2133 MHz RAM.

The results for this analysis are displayed in Figures 4.3-4.5. Figure 4.3 displays the computed subspace reachable set in the x, y, J_1 subspace at the final time horizon of 5 days. This represents the set of possible combinations of final positions as a function of J_1 or ΔV cost. When there is no thrusting at all, the J_1 cost is maximized, ΔV cost is minimized, and the spacecraft doesn't move since L1 is an equilibrium point of the cislunar

system of Eq. (4.48). This location corresponds to the "top" of this subspace reachable set, denoted by the black circle. Intuitively, as the thruster provides more acceleration to this system, the set of reachable final states also increases.

Figures 4.4 - 4.5 display the same results but displays the optimal state space trajectories at the final time horizon of 5 days. In these views, the different levels of control effort or ΔV are given by horizontal planes. Figure 4.5 shows that optimal trajectories tend to begin with a period of thrusting followed by a period of non-thrusting (coasting). These mirror the classical results derived from primer vector theory which shows that the minimum-fuel consumption optimal control policy is based on periods of coasting and thrusting [108].

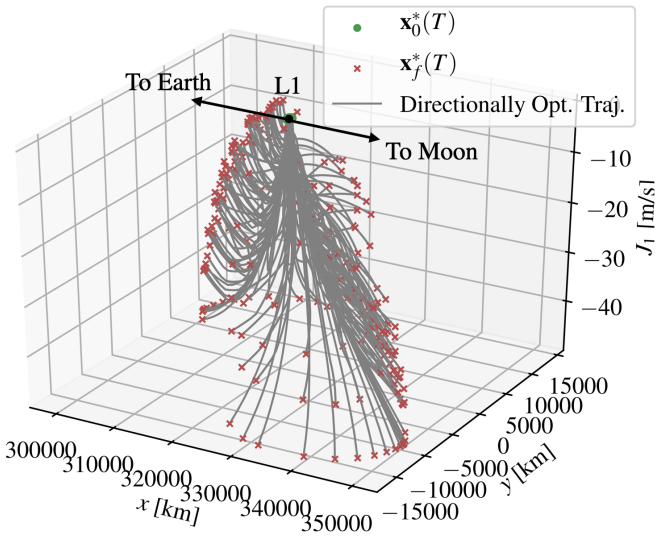


Figure 4.4: Optimal trajectories at final time horizon in x, y, J_1 space.

Without acceleration input from the thruster, it can be shown that a constant of motion, called the Jacobi constant, exists for this problem [80]. For the planar system given in Eq. (4.48), the Jacobi constant is expressed as

$$C = (x^2 + y^2) + 2\frac{1-\mu}{\rho_1} + 2\frac{\mu}{\rho_2} - (\mathring{x}^2 + \mathring{y}^2) \tag{4.54}$$

The Jacobi constant is a relative energy measure that is used to study feasible trajectories

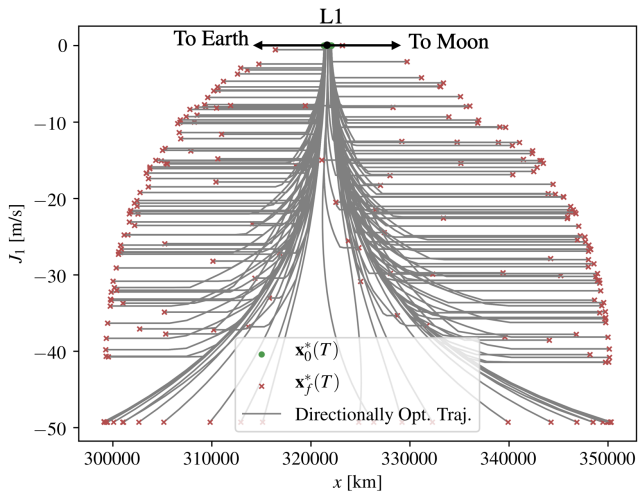


Figure 4.5: Optimal trajectories at final time horizon in x, J_1 space.

of this system given initial conditions. The smaller C is, the more relative energy the spacecraft has and vice-versa. For a given value of C determined by initial conditions and no control input, a zero-velocity surface in x, y can be computed denoting the extremal points on the trajectory for all time. Figure 4.6 displays the feasible zero-velocity surfaces both before and after the reachability scenario. These contour values denote the range of possible Jacobi constant values. Figure 4.6 also shows that the L2 Lagrange point becomes is reachable after the 5 day time horizon.

Figure 4.7 displays the how the position subspace reachable set changes with J_1 cost or ΔV usage. This type of analysis can also be used to infer solutions of two-point boundary value problems that originate from trajectory optimization problems. Suppose there is a desired position, denoted by $x_d = 2000\text{km}, y_d = -2000\text{km}$, that the spacecraft is tasked to reach within the 5 day time horizon. Traditional trajectory optimization problems typically require the specification of separate weights between the time cost and the control effort cost. The corresponding single objective optimal control problem is then solved to generate both the optimal control policy and state trajectory. However, if the tradeoffs between minimum time and minimum control effort are desired, this process needs to be repeated with varying weights between the two objectives, analogous to the discussed weighted sum

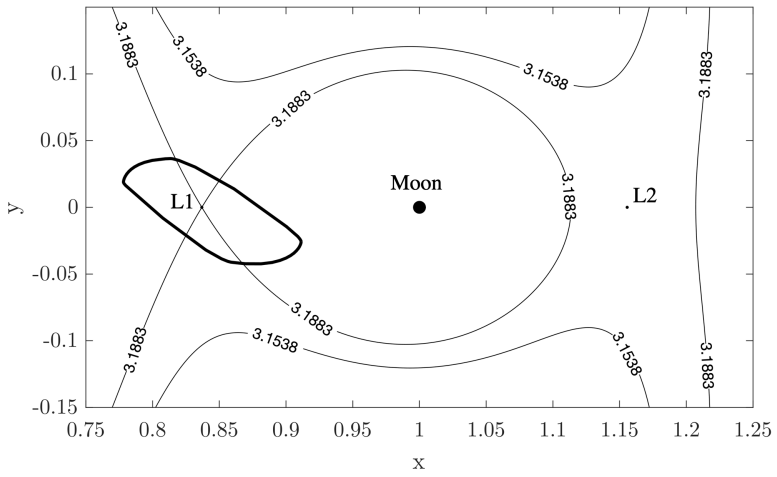


Figure 4.6: Feasible zero-velocity surface contours denoting Jacobi constant value before and after 5 day period. Position subspace reachable set at 5 day horizon is also shown. Nondimensional units are used. Moon to scale.

technique in optimization.

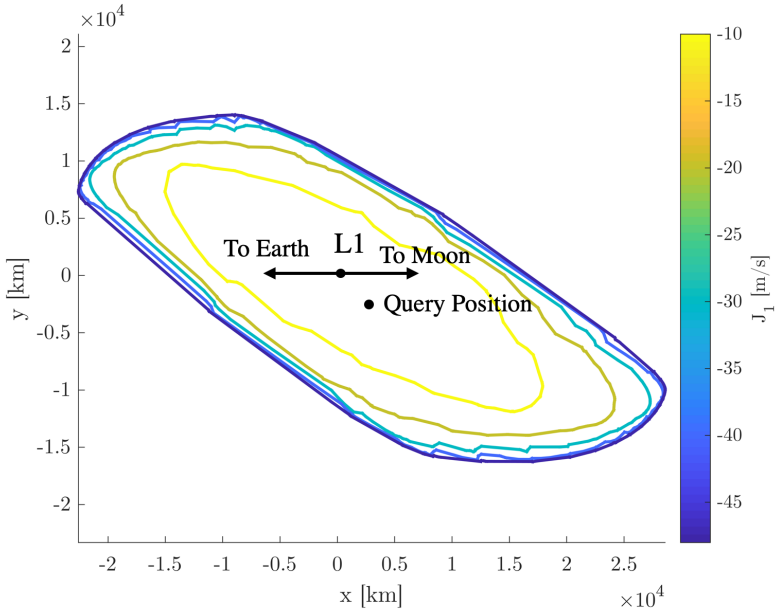


Figure 4.7: ΔV Contours for cislunar trajectory optimization problem. The contour levels are -10, -20, -30, -40, -49 m/s. The desired position in the trajectory optimization demonstration is also shown.

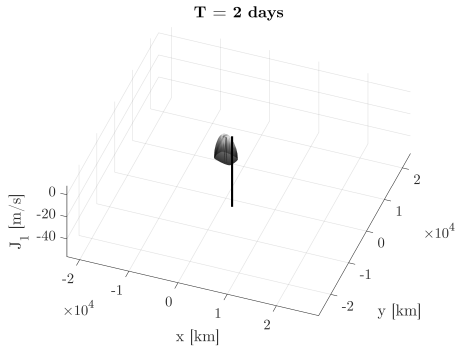
This type of analysis may be performed using the joint reachability and multi-objective optimization formulation presented in this thesis. The benefit to an analysis in this formula-

tion is that the minimum-time and minimum control effort trajectories are computed for all reachable states at the specified time horizon. The minimum-time solution is determined by observing when the desired state first intersects with the reachable set. If the desired state is instead within the interior of the reachable set at a particular time, there exists a tradeoff between minimum-time and minimum control effort trajectories that terminate at the desired location.

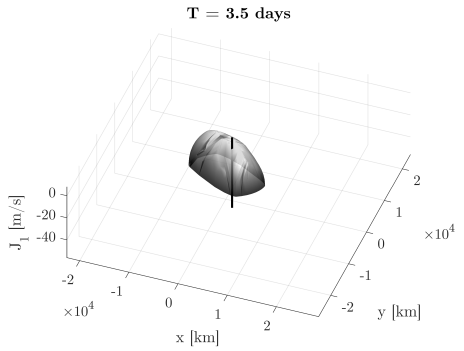
In this demonstration as control effort cost is one of the objectives, one can use the extremal set to provide minimum control effort trajectories for positions that are within the position subspace reachable set. For a desired position within the position subspace reachable set, one may project this state onto the extremal set to get the minimum control effort required to reach the desired position. This process is demonstrated in Figure 4.8 where the x, y, J_1 subspace reachable sets are shown as the time horizon increases. The vertical blue line specifies the desired position of this trajectory optimization problem, $x_d = 2000\text{km}, y_d = -2000\text{km}$. After one day has passed, it is not possible to achieve the desired position because it lies outside of the reachable set. After about 2.3 days has passed, the desired position first intersects with the reachable set, corresponding to the minimum time solution. After this time horizon, the desired position remains within the reachable set and the minimum control effort cost can be recovered by finding the intersection of the desired position state with the extremal set.

This process can be repeated at different reachability time horizons in order to compute the Pareto-optimal curve for minimum-time and minimum control effort, as shown in Figure 4.9. This type of solution not only provides the set of efficient tradeoffs between time and control cost, it also provides the sensitivities between these two costs. For example, the required ΔV substantially drops near the minimum time solution of about 2.3 days while the required ΔV drops much less as the time horizon increases.

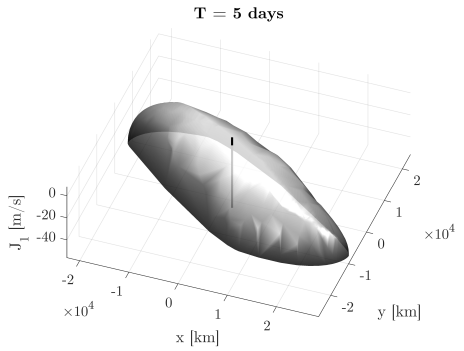
While this type of analysis is demonstrated using the desired position of $x_d = 2000\text{km}, y_d =$



(a) x, y, J_1 subspace reachable set at 2 days



(b) x, y, J_1 subspace reachable set at 3.5 days



(c) x, y, J_1 subspace reachable set at 5 days

Figure 4.8: x, y, J_1 subspace reachable sets over time horizon with black line displaying the desired position (x, y) =(2000 km , -2000 km) relative to L1 Lagrange point

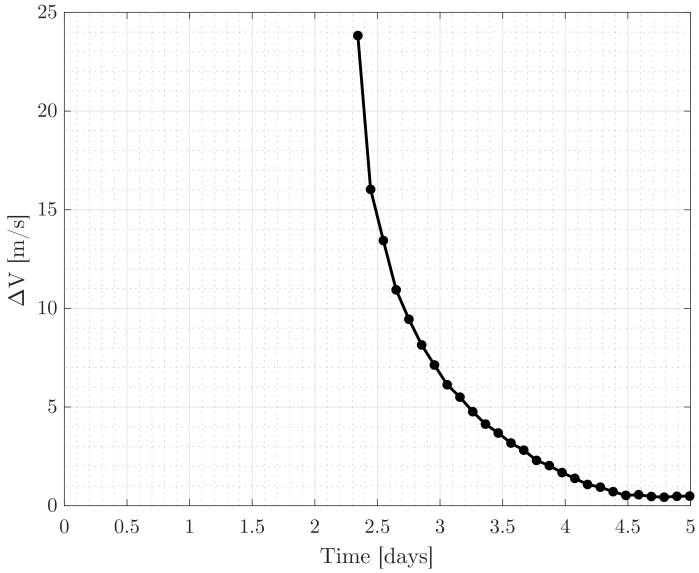


Figure 4.9: Minimum-ΔV and minimum-time Pareto-optimal curve for the desired position (x, y) =(2000 km , -2000 km) relative to L1 Lagrange point

-2000km, this can be repeated for any other desired position that becomes reachable within this time horizon of 5 days. Moreover with to the joint formulation, more objective functions and design variables may be added to this analysis to generate extremal sets and efficient tradeoffs of more complex problems.

4.7 Conclusions

This chapter describes analytical relationships between optimal control, reachability, and multi-objective optimization theory. The first-order necessary conditions of optimality are presented for each problem type and the similarities constructs are discussed.

A joint reachability and multi-objective optimization formulation is presented that combines dynamic states, objective functions, and design parameters into a single analysis framework. Through the joint fomulation GIP HJB PDE, extremals sets can be computed representing the simultaneous set of feasible objective functions, design variables and dynamic states. By simplifying the joint formulation GIP HJB PDE, it is shown that problems

specific to optimal control, reachability, and multi-objective optimization fields can be recovered.

The reachability problem is cast as a multi-objective optimization problem where the objective functions are replaced by the state trajectory flow functions and the design variables are replaced by the initial states and/or the initial costates. A sample formulation is presented and numerical solution methods within the multi-objective optimization field are discussed. Additionally, using homotopy principles, the multi-objective optimization problem is cast as a reachability problem by introducing homotopy maps. Numerical solution techniques characteristic to reachability analyses are reviewed.

These concepts are first demonstrated on a classical, biccriterial multi-objective optimization problem. Extremal points that sample the boundary of the feasible objective function space are computed using the SCoRe algorithm. The second demonstration solves a multi-objective optimal control problem of maneuverability of a spacecraft near the Earth-Moon L1 Lagrange point. A further analysis of these results leads to methods for computing solutions to two-point boundary value problems and multi-objective trajectory optimization problems.

CHAPTER 5

REACHABILITY TOOLBOX COMPARISON

One of the most general techniques for computing reachable sets is by using the level sets (LS). Mitchell created a toolbox in MATLAB to compute viscosity solutions of the HJB PDE over state space using level set techniques [7]. This toolbox is commonly used throughout the aerospace community due to its accurate results and system generality. However, the computation time required with this toolbox increases drastically when the subspace of interest is high dimensional ($n > 4$) and when there is a large state space scale. Furthermore, because the overall analysis space requires a grid, the size of the reachable set must be known *a priori*.

More recently, other reachability analyses methods have shown great computational efficiency by computing reachable sets of hundreds of states. However, these methods generally rely on overapproximating the reachable set using conservative linear approximations of the system dynamics. In this thesis, the CORA toolbox is used [13] as it is implemented in MATLAB and is commonly used in high-dimensional reachability analyses [128, 82].

As previously discussed, the methods presented in this thesis share many similarities with both of these families of reachability computation. This technique uses optimal control and the accompanying necessary conditions of optimality to quantify and ensure accuracy. Furthermore, by using a sample-based technique and support functions, samples of the reachable set boundary can be efficiently computed, even if the state dimension is large. Consequently, the methods presented in this thesis aim to be a compromise between the accuracy and system generality of the HJB methods with the speed and dimensional scalability of the geometric set-based methods.

To highlight some of the advantages of this technique, the following reachability analysis is performed using the Level Set (LS) toolbox, CORA toolbox, and a Python implementation of the presented methods. Our Python implementation will be denoted as Sampled Continuation Reachabbility (SCoRe) from this point on. As all of these implementations have capabilities of specifying desired accuracy, the following reachability analysis is conducted over multiple accuracy settings.

Dubin's Car Model

The dynamic system of interest for this comparison is an extension of the classical Dubin's car model. This scenario is used because it is a nonlinear system, it is a commonly used problem scenario in optimal control, and is three-dimensional. The dynamics model is given by

$$\begin{aligned} \dot{x} &= u_1 \sin(\theta) \\ \dot{y} &= u_1 \cos(\theta) \\ \dot{\theta} &= u_2 \end{aligned} \tag{5.1}$$

where $\|\mathbf{u}\|_2 \leq 1$ and the set of initial states is given by

$$100(\mathbf{x} - \mathbf{x}_c)^T(\mathbf{x} - \mathbf{x}_c) \leq 1$$

where $\mathbf{x}_c^T = [0, 0, \pi/2]$ and the reachable set is computed after $\pi/2$ seconds.

To compute the reachable set using the LS toolbox, a $51 \times 51 \times 51$ grid of states from -2 to 2 for x, y and -0.5 to 3.5 for θ. The desired range for the grid was found by trial-and-error and finding a box region that completely contained the reachable set. The LS toolbox allows the user to control the order of the approximations using simple accuracy settings of `` `low' ``, `` `medium' ``, `` `high' ``, and `'veryHigh'`.

To compute the reachable set using the CORA toolbox, a zonotopic approximation to both

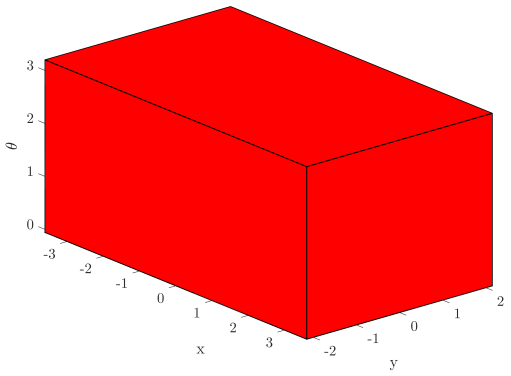
the feasible control set and the initial state set had to be generated. For the initial state set, the zonotope representation of a truncated small rhombicosidodecahedron was used to closely approximate a sphere [129] of the appropriate size. To approximate the feasible control set, 30 unit circle vectors were used as the generators to the zonotope then scaled to the appropriate size. The main parameters that affect the accuracy of the solution are the `options.maxError` variable and the time step size.

For our python implementation, 160 samples are generated and a full three-dimensional reachable set is computed. The accuracy is controlled by specifying the relative and absolute integration tolerances for the continuation method differential equations. For simplicity these values are set to the same value for every reachable set computation. Additionally, the Newton's method corrective steps are disabled to measure accuracy due to the numerical continuation method itself.

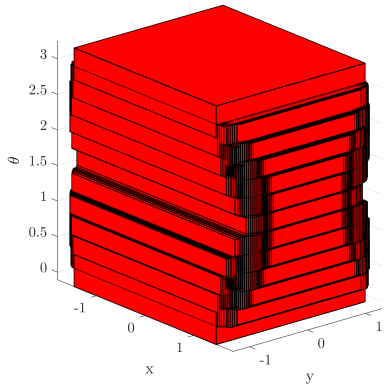
The truth model for the reachable set was generated by the LS toolbox with a $101 \times 101 \times 101$ grid of states with the `'veryHigh'` accuracy setting. This truth model reachable set took 405 seconds to compute.

Figure 5.2 displays the results of projecting the computed reachable sets onto the different subspaces. CORA-TS denotes the solution with large number of time steps and CORA-ME denotes the solution with low value of `options.maxError`. The LS method accuracy was set to `'veryHigh'`. However, for the SCoRe method presented in this thesis, the accuracy was set using the integration tolerances of 1e-3. As shown, the LS and Brew methods produced similar results while the CORA results form overapproximations to the reachable set.

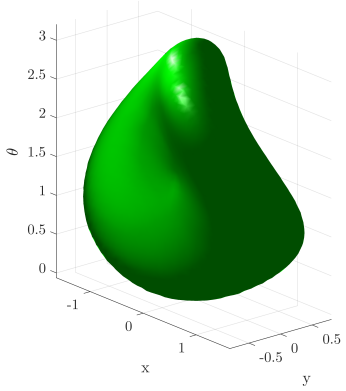
The regions in Figure 5.2 are generated by connecting a line between the outermost samples once they are projected onto a subspace. It should be noted that the true reachable set in this example is non-convex. The SCoRe method still computes samples on the reachable set that constitute its convex hull, by construction. Consequently, there are no guarantees



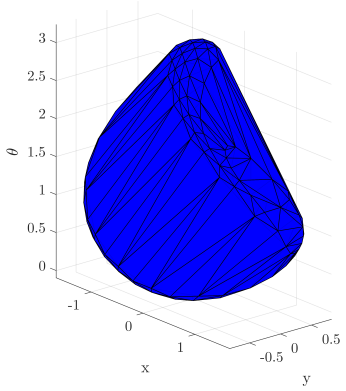
(a) Reachable set computation from CORA toolbox with 1280 time steps and `options.maxError`=1e20 accuracy setting



(b) Reachable set computation from CORA toolbox with 50 time steps and `options.maxError`=0.3

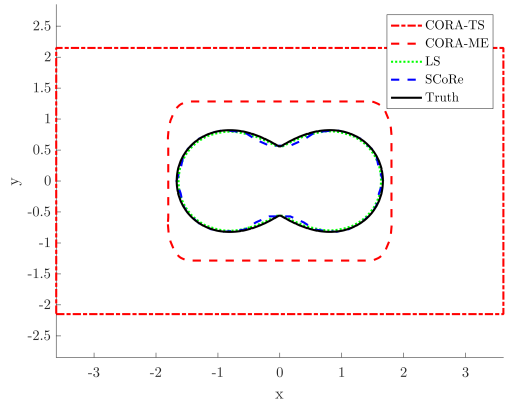


(c) Reachable set computation from LS toolbox with `veryHigh` accuracy setting

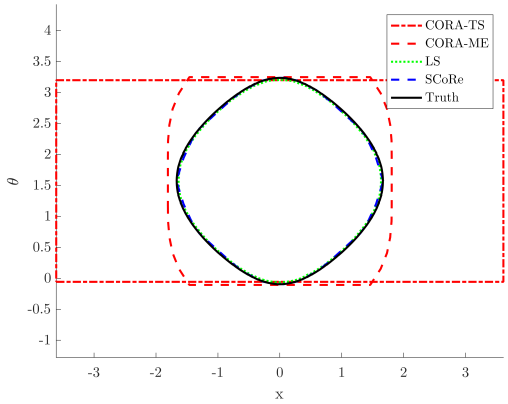


(d) Reachable set computation from SCoRe toolbox with 1e-3 accuracy setting

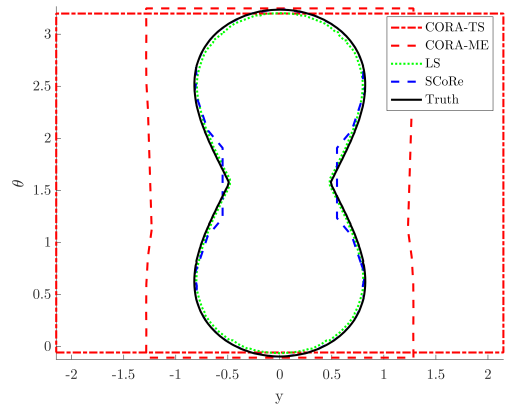
Figure 5.1: Reachable set comparison between the three different reachability algorithms



(a) Projection of reachable set on the x, y subspace



(b) Projection of reachable set on the x, θ subspace



(c) Projection of reachable set on the y, θ subspace

Figure 5.2: Reachable set projection comparison between the three different reachability algorithms

that the subspace projection of the convex hull samples will sufficiently sample the true reach set projection.

As previously discussed, the satisfaction of the first order necessary conditions of optimality can be used an accuracy metric for a reachable set computation. However, this requires surface tangent or costate information to evaluate. The SCoRe method and the LS toolbox both provide surface tangent and costate information but the CORA method does not explicitly supply this information.

For a fair comparison, another acccuracy measure is used that solely uses sample information. This alternate measure can be derived from the Euclidean distance between the reachable set samples with the true reachable set model. Denote the projection distance of a single sample point $\mathbf{x}$ to the truth reachable set $\mathcal{R}$ as

$$\begin{aligned} d_{\mathcal{R}}(\mathbf{x}) = \min_{\mathbf{x}_{\text{proj}}} \|\mathbf{x}_{\text{proj}} - \mathbf{x}\|_2 \\ s.t. \quad \mathbf{x}_{\text{proj}} \in \partial\mathcal{R} \end{aligned} \tag{5.2}$$

This projection distance is used as a measure of accuracy since it quantifies deviations from the true reachable set. When evaluated for every sample of the reachable set computation, the following state distance set is created $d_{\mathcal{R}}(\mathcal{X}) = \{d_{\mathcal{R}}(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}$ where $\mathcal{X} \subset \mathbb{R}^n$ denotes the set of samples for a particular reachable set computation. Figure 5.3 in shows the results of the state distance set in cumulative density form.

Figure 5.3 shows that the samples from the SCoRe method are the most accurate compared to the highest accuracy settings of the other two methods. This is because the presented method computes point solutions that satisfy the necessary conditions of optimality without making assumptions on the form of the reachable set, using approximations for the nonlinear system dynamics. The CORA toolbox overapproximates the reachable set due to approximations formed for the feasible control set, intial state set, nonlinear dynamics, and set representation. The LS toolbox underapproximates the reachable set due to insufficient

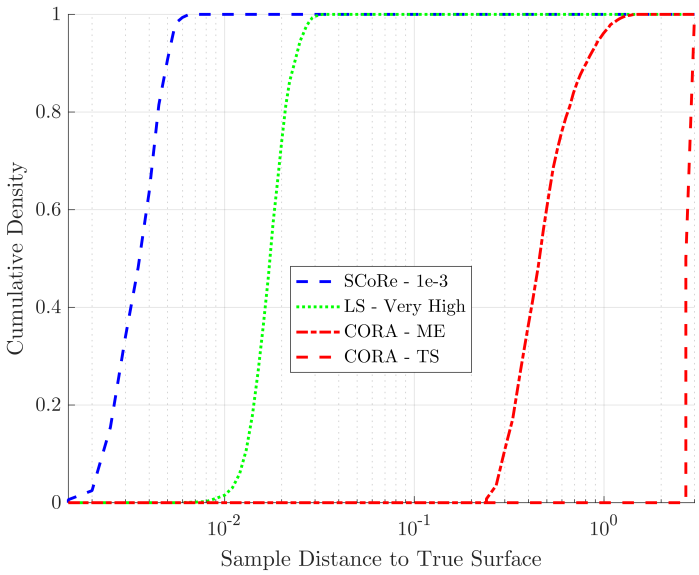


Figure 5.3: Cumulative density functions for each state distance set $d_{\mathcal{R}}(\mathcal{X})$

grid density and the addition of viscosity terms in the propagation of the level sets.

For the next comparison, the accuracy settings of each method are varied in order to gather information on computation time and accuracy tradeoffs. With each accuracy setting, the average value of the state distance set $d_{\mathcal{R}}(\mathcal{X})$ and the computation time is recorded. For the Python implementation, the three accuracy settings are relative and absolute integration tolerances of 1e-3, 1e-6, and 1e-9. For the LS toolbox, the accuracy settings are given by `` `medium' ``, `` `high' ``, and `'veryHigh'`. For the CORA toolbox, the accuracy settings are given by adjusting the `options.maxError` value to 1.0, 0.5, and 0.3 and the number of time steps to 5, 80, and 1280.

All of the computations were completed on a single-core of a MacBook Pro 2.3 GHz Intel Core i5 processor with 8 GB 2133 MHz RAM. The LS and CORA toolboxes are both implemented in MATLAB while the SCoRe method is implemented in Python.

The goal of any reachability analysis algorithm is to compute reachable volumes with low error in little time. However, as with many examples in numerical computing, a tradeoff exists between the accuracy of the algoritm and the time it takes to run the algorithm. Figure

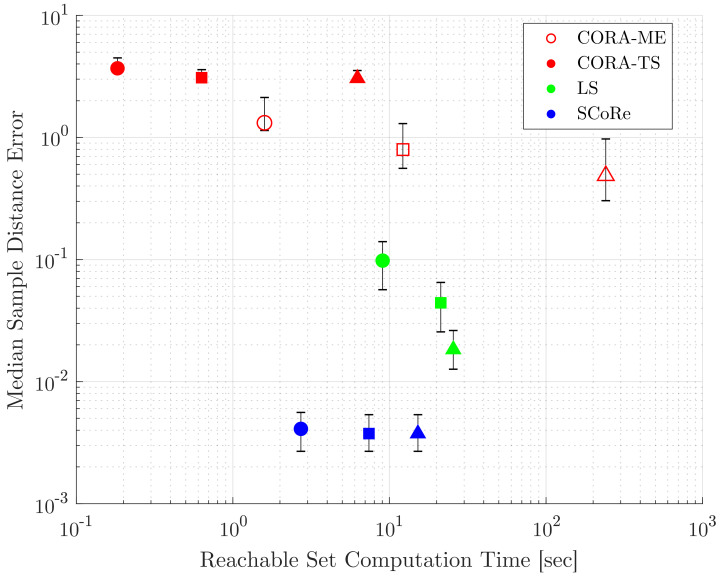


Figure 5.4: Computation time and reachable set sample accuracy tradeoffs for each method along with error bars denoting 5% and 95% percentiles. The lowest, medium, and highest accuracy settings used in the comparison are denoted by ●, ■, and ▲, respectively.

5.4 shows each of the reachable set algorithms at the specified accuracy levels along with error bars denoting 5% and 95% percentiles for the sample distances to the true reachable set model. It should be noted that the error bars are also logarithmically scaled.

For this problem, the CORA toolbox struggled in terms of both computation time and accuracy.

For this problem, the CORA toolbox is able to compute the reachable sets in the least amount of time. However, in order to achieve a large improvement in the solution accuracy, a large computational cost is incurred. This is mostly due to the overapproximate dynamics and discretization of the initial state and feasible control set. In general, the CORA toolbox performs well in analyses of dynamic systems with rectangular sets of initial state variation and feasible control set. Furthermore, the CORA toolbox scales well with increasing state dimension.

As expected, the LS toolbox outperformed the CORA toolbox in terms of accuracy. For the LS toolbox, Figure 5.4 shows the overall tradeoff between required computation time and

desired accuracy - for improvements in the solution accuracy, the computational time will be lengthened. Another method for improving the accuracy of the LS method is to refine the state space grid. However, the increased computational load due to the grid refinement scales exponentially. It should be noted that the LS toolbox does compute the full, non-convex reachable set while the other methods do not capture the non-convex regions of the reachable set.

The SCoRe algorithm performed the best in terms of sample accuracy, mirroring the results of Figure 5.3. Additionally, the computation times are on the same or less order of magnitude as the "medium" accuracy settings of the other methods. However, this method only computes samples that lie on the convex hull of the true reachable set. Consequently, a convex representation of the true reachable set is computed as opposed to the non-convex reachable set computed from the LS that better matches the actual shape of the reachable set. The method doesn't increase much in accuracy as the accuracy settings are varied and the required computation time increases. This asymptotic error is mostly due to the definition of the reachable set truth model using the LS toolbox and to numerical errors associated with computing $d_{\mathcal{R}}(\mathcal{X})$. This asymptotic error may be reduced by including the corrective Newton's steps in the presented method or by further refining the reachable set truth model.

6-Dimensional Relative Motion Problem

The second problem scenario is given by the 6-dimensional reachability analysis of an object in Keplerian orbit about the Earth. The exact nonlinear relative equations of motion

for an object about a given arbitrary reference orbit $\mathbf{x}_r(t)$ are

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ 2\dot{f}_r\left(\dot{y} - y\frac{\dot{r}_r}{r_r}\right) + x\dot{f}_r^2 + \frac{\mu}{r_r^2} - \frac{\mu}{r^3}(r_r + x) + u_x \\ -2\dot{f}_r\left(\dot{x} - x\frac{\dot{r}_r}{r_r}\right) + y\dot{f}_r^2 - \frac{\mu}{r^3}y + u_y \\ -\frac{\mu}{r^3}z + u_z \end{bmatrix} \tag{5.3}$$

where the true anomaly rate $(\dot{f}_r)$, reference radius (r_r), and reference radius time derivative $(\dot{r}_r)$ can be directly computed using Keplerian dynamics and the inertial radius of the spacecraft is r, defined as $r = \sqrt{(r_r + x)^2 + y^2 + z^2}$ [80]. These equations of motion represent the relative motion between an object and another reference object in a reference orbit. The dynamics are expressed in a rotating Hill frame, where the radial axis (x) points from the center of the Earth to the reference object and the along-track axis (y) is defined as perpendicular to the radial vector and is positive in the direction of the reference orbit velocity.

The discussed approach is demonstrated on the case of a spacecraft in an eccentric geostationary transfer orbit (GTO) with maximum thrust constraints. For this demonstration, the initial condition set is given by

$$g(\mathbf{x}_0) = \mathbf{x}_0^T E \mathbf{x}_0 \leq 1 \tag{5.4}$$

where $E =$ diag(10, 10, 10, 0.1, 0.1, 0.1) and the feasible control set as

$$\|\tilde{\mathbf{u}}\|_5 \leq 1 \tag{5.5}$$

where $u_i =$ 1e-6 $\tilde{u}_i, i = 1, 2, 3$. The nominal GTO orbit has a periapsis radius of 7000

km and an apoapsis radius at GEO with 42164 km. In this demonstration, the position subspace reachable set (x, y, z) at the apoapsis of the nominal orbit is computed using the CORA and SCoRe algorithms.

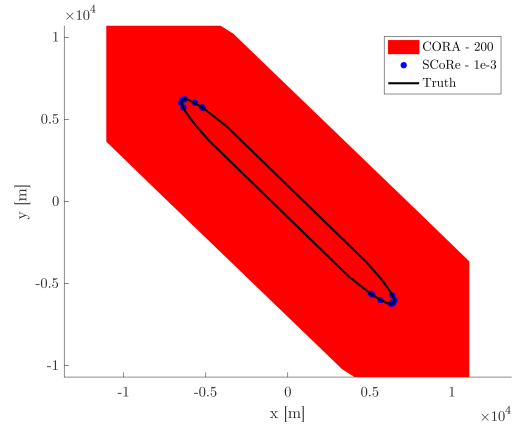
While the full state space dimension for this problem is 6, the position subspace reachable set is 3-dimensional. Due to the curse of dimensionality, this prevents a reachability analysis from be conducted using the LS. A projective LS technique has been studied that allows for subspace reachable sets to be evaluated [10] in the LS framework. Similar to the SCoRe method, computing subspace reachable sets incur computational costs in the subspace of interest as opposed to the full state space. At the time of the writing, the authors don't have an implementation of the projective LS method. Consequently, the LS method will not be evaluated for this problem comparison.

The truth model for the reachable set was generated by the SCoRe algorithm with 162 particles and integration tolerances of 1e-9. To ensure the accuracy of this truth model, Newton's root-finding was performed on each particle individually to a convergence tolerance of $\|\mathbf{F}(\mathbf{z})\| \leq$ 1e-9.

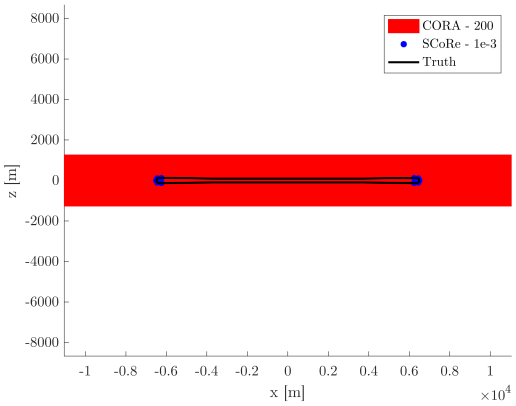
This problem scenario uses the same CORA algorithm parameters used for the Dubin's car model previously discussed. The variable that parametrizes the accuracy in this case for the CORA algorithm is given by the number of time steps. The chosen values for the number of time steps are 50, 100, 150, and 200.

Figure 5.5 displays the results of projecting the computed reachable sets onto the different subspaces. The CORA reachable set is smaller in the x, y subspace than the truth. However, the z-direction yielded accurate results. This is most likely due to the coupled nonlinear dynamics of the x, y motion. The motion in the z component is lightly coupled with the other two components.

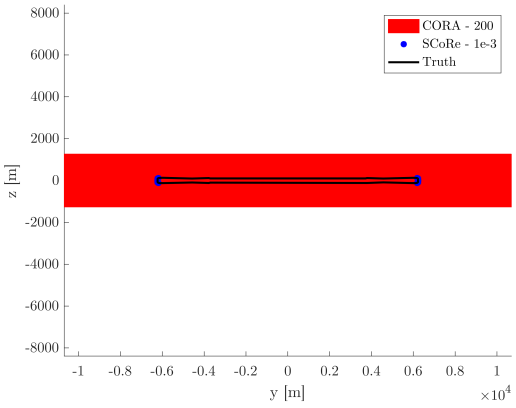
Similarly to the previous Dubin's car comparison, the accuracy settings of each method are varied in order to gather information on computation time and accuracy tradeoffs. With



(a) Projection of reachable set on the x, y subspace



(b) Projection of reachable set on the x, z subspace



(c) Projection of reachable set on the y, z subspace

Figure 5.5: Reachable set projection comparison between CORA and SCoRe reachability algorithms

each accuracy setting, the average value of the state distance set $d_{\mathcal{R}}(\mathcal{X})$ and the computation time is recorded. For the Python implementation, the three accuracy settings are relative and absolute integration tolerances of 1e-3, 1e-6, and 1e-9. For the CORA toolbox, the accuracy settings are given by adjusting the `options.maxError` value from 1e-4 to 0.4e-4 in 0.1e-4 increments.

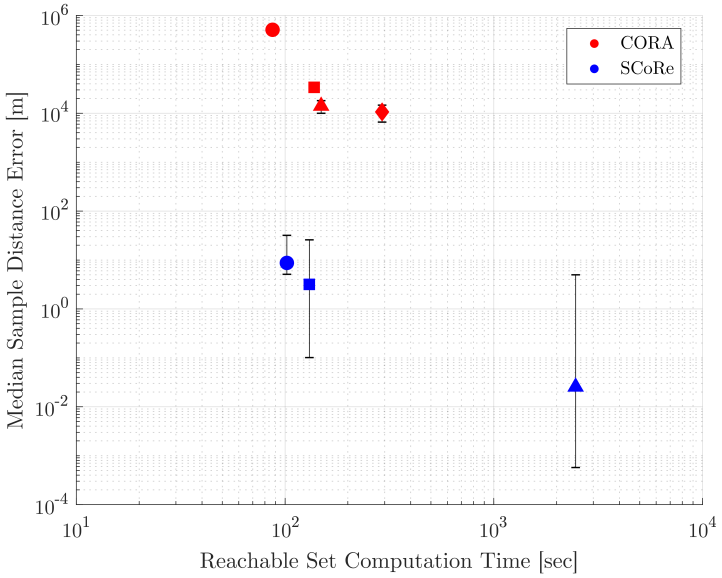


Figure 5.6: Computation time and reachable set sample accuracy tradeoffs for each method along with error bars denoting 5% and 95% percentiles. The markers denoting the accuracy settings used in the comparison are denoted by ●, ■, and ▲ in the direction of increasing accuracy.

For this problem, the CORA toolbox produced reachable sets in a faster amount of time, but the solution accuracies were orders of magnitude worse than the SCoRe solutions. Furthermore, the CORA toolbox computes the full 6-dimensional reachable set while the SCoRe computes the 3-dimensional position subspace reachable set. The SCoRe algorithm generates accurate results but took comparatively longer than the CORA when computing the position subspace reachable set. The required computation time for the SCoRe algorithm doesn't scale as well as the CORA method when considering the overall time horizon. However, performance improvements can be made in the SCoRe algorithm by performing corrective Newton's steps throughout the continuation method. Further performance im-

provements can be achieved if this algorithm is implemented on a parallelized computing platform.

This comparison between the SCoRe method alongside the LS and CORA toolboxes is by no means comprehensive. There are many other dynamic systems, reachability analsysis scenarios, and algorithm parameters that could lead to different results. Furthermore, this comparison doesn't explore all the available reachable set analysis methods. This comparison is not meant to provide a conclusive result in terms of which algorithm to use in every situation, it is meant to highlight some of the differences between the methods. For a given reachability analysis, the choice of reachability algorithm depends heavily on the type of dynamic system, restrictions on dynamic states and control input, desired accuracy of the solution, and the computational requirements.

CHAPTER 6

CONCLUSIONS

The concept of reachability directly addresses issues in system level autonomy, system safety verification, system fault detection, and system design. Once computed, reachable volumes can be used to make guarantees of performance of a given dynamic system. Such guarantees are critical because they allow one to confidently predict where the system state may be in the future or what previous states may have generated the current one. Whether the system is a near-Earth asteroid, fighter jet, chemical process, or a hurricane, predictions from reachability analyses can help to better inform decision makers.

Numerous analytical, computational, and software tools have been developed within the past decade to compute reachable sets, mainly for the task of continuous and hybrid system verification analysis. The challenges in developing reachability algorithms include generality of system dynamics, representation of reachable volumes, and computational tractability. Generally, these algorithms make performance trade-offs between each of the above stated challenges.

Chapter 2 provides fundamental results in the application of numerical continuation methods to reachability analyses. In this chapter, the formulation for using numerical continuation methods to compute samples of forwards and backwards reachable sets and tubes is defined. Numerical techniques to suppress error in the continuation method are introduced including root-finding corrective steps, matrix equilibriation, and psuedo-arclength continuation. While current methods are limited to ellipsoidal feasible control sets and initial condition sets, Chapter 2 generalizes this concept to affine transformations of unit balls of normed vector spaces. This allows for analytic minimum-time optimal control policies for

control affine systems. Unions of convex initial condition sets are also discussed in this chapter, enabling the computation of reachable volumes from non-convex initial condition sets. Importantly, this study extends the current state-of-the-art of reachability theory using numerical continuation methods and lays the foundation for the remainder of the thesis.

Chapter 3 provides analytic results toward the distribution of samples along the reachable set boundary as well as numerical methods to achieve the desired sampling uniformity or spatial resolution. In this chapter, it is proven that curvature-based sampling occurs when using support-function-based reachability objective functions and uniform sampling of unit vector search directions. If uniform reachable surface coverage is desired, Chapter 3 outlines the necessary conditions for uniform sample coverage based on a decentralized graph theory and analytical mechanics approach. Additional techniques for updating the distribution of point solutions based on spawning new samples are also introduced. Overall, this study develops numerical methods for achieving a desired surface sampling and introduces the concept of numerical continuation along the reachable set boundary to locally explore the extremal surface.

Chapter 4 explores analytical connections between reachability theory and multi-objective optimization. In this chapter, a joint reachability and multi-objective optimization formulation is presented. This allows for the computation of joint extremal sets containing information of dynamic states, objective functions, and design parameters. Moreover, this joint formulation generalizes to reproduce results from optimal control, reachability, and multi-objective optimization problems. Demonstrations of the joint formulation include the ability to cast reachability problems as multi-objective optimization problems and vice-versa. Emphatically, the connections developed in Chapter 4 help bridge the gap between the fields of multi-objective optimization and reachability. Moreover, unification of these previously distinct areas enable cross-pollination of both theory and numerical methods.

Chapter 5 presents a numerical comparison between the implementation of the theory pre-

sented in this thesis with two commonly used reachability algorithms. In this chapter, reachability solutions using the Sampled Continuation Reachability (SCoRe) algorithm, Level Set Toolbox, and the CORA toolbox are compared when applied to the problems of a classical Dubin's car and a spacecraft in relative motion about a nominal orbit. The results presented in this chapter highlight advantages of the SCoRe methods such as maintaining a high level of accuracy with tractable computation times.

The problem of efficiently computing reachable volumes has been studied for decades and much work remains in order to generate accurate results that can be used to make time-critical decisions. The work in this thesis contributes to this problem by introducing computational techniques based on numerical continuation to the reachability field. Moreover, through the described connections between multi-objective optimization theory, a large number of analytic and numerical references are introduced to the reachability field. The demonstrations presented in this thesis show promise for the use of numerical continuation in the development of computationally efficient algorithms in reachability applications.

Appendices

APPENDIX A

DERIVATIONS

A.1 Hamilton Jacobi Bellman PDE

The Optimal Control Problem is formally stated as

$$\begin{aligned} \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\int_{t_0}^{t_f}\mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)d\tau + V(\boldsymbol{x}_f,t_f)\right] \\ \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t) \\ \boldsymbol{h}(\boldsymbol{x},t) \leq \boldsymbol{0} \\ \boldsymbol{g}(\boldsymbol{x}_0,t_0,\boldsymbol{x}_f,t_f) = \boldsymbol{0} \end{aligned} \tag{A.1}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the state, $\boldsymbol{u} \in \mathbb{R}^m$ is the control input, $t \in [t_0, t_f]$ is time, $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ is the trajectory Lagrangian, $V : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is the terminal performance function, $\boldsymbol{f} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n$ captures the system differential equations, $\boldsymbol{h} : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^q$ defines trajectory inequality constraints, $\boldsymbol{g} : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^v$ expresses boundary conditions, $U \subseteq \mathbb{R}^m$ defines the set of admissible controls, and the 'opt' argument is understood to denote an arbitrary optimization depending on the application, i.e. 'sup', 'inf', 'min', or 'max'. In addition, for reachability problems in general there are no trajectory inequality constraints placed on the state. Inequality constraints may be considered using the inclusion of slack variables or by augmenting the optimal control objective function with penalty functions.

The performance index in Eq. (A.1) can be restated in the Dynamic Programming form as

$$V(\boldsymbol{x}_0,t_0) = \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\int_{t_0}^{t_f}\mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)d\tau + V(\boldsymbol{x}_f,t_f)\right] \tag{A.2}$$

More generally, this is the Principle of Optimality expressed as

$$V(\boldsymbol{x}(t),t) = \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\int_t^{t+\Delta t} \mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)d\tau + V(\boldsymbol{x}(t+\Delta t),t+\Delta t)\right] \tag{A.3}$$

which relates the current optimal cost-to-go ($V(\boldsymbol{x},t)$) with the optimal cost-to-go at a future time ($V(\boldsymbol{x}(t+\Delta t),t+\Delta t)$). Each term in Eq. (A.3) is now expanded using a Taylor series.

$$\boldsymbol{x}(t+\Delta t) = \boldsymbol{x}(t) + \boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t)\Delta t + o(\Delta t) \tag{A.4}$$

$$\begin{aligned} V(\boldsymbol{x}(t+\Delta t),t+\Delta t) &= V(\boldsymbol{x}(t),t) + \frac{\partial V}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),t)[\boldsymbol{x}(t+\Delta t)-\boldsymbol{x}(t)]\Delta t + \frac{\partial V}{\partial t}(\boldsymbol{x}(t),t)\Delta t + o(\Delta t) \\ &= V(\boldsymbol{x}(t),t) + \frac{\partial V}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),t)\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t)\Delta t + \frac{\partial V}{\partial t}(\boldsymbol{x}(t),t)\Delta t + o(\Delta t) \end{aligned} \tag{A.5}$$

$$\int_t^{t+\Delta t} \mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)d\tau = \mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)\Delta t + o(\Delta t) \tag{A.6}$$

Substituting these results back into the Principle of Optimality generates

$$V(\boldsymbol{x}(t),t) = \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)\Delta t + V(\boldsymbol{x}(t),t) + \frac{\partial V}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),t)\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t)\Delta t + \frac{\partial V}{\partial t}(\boldsymbol{x}(t),t)\Delta t + o(\Delta t)\right] \tag{A.7}$$

Observing that $V(\boldsymbol{x}(t),t)$ and $\frac{\partial V}{\partial t}(\boldsymbol{x}(t),t)\Delta t$ do not depend on $\boldsymbol{u}$ yields

$$-\frac{\partial V}{\partial t}(\boldsymbol{x}(t),t)\Delta t = \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\mathcal{L}(\boldsymbol{x}(\tau),\boldsymbol{u}(\tau),\tau)\Delta t + \frac{\partial V}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),t)\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t)\Delta t + o(\Delta t)\right] \tag{A.8}$$

Dividing by Δt and letting $\Delta t \to 0$ yields the traditional Hamilton-Jacobi-Bellman PDE

as follows

$$-\frac{\partial V}{\partial t}(\boldsymbol{x}(t),t) = \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\mathcal{L}(\boldsymbol{x},\boldsymbol{u},t) + \frac{\partial V}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),t)\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t)\right] \tag{A.9}$$

Traditionally, the Hamiltonian $\mathcal{H}(\boldsymbol{x},\boldsymbol{u},\boldsymbol{p},t)$ is defined as

$$\mathcal{H}(\boldsymbol{x},\boldsymbol{u},\boldsymbol{p},t) = \mathcal{L}(\boldsymbol{x},\boldsymbol{u},t) + \boldsymbol{p}^T\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t) \tag{A.10}$$

where $\boldsymbol{p}$ is the costate variable corresponding to the state $\boldsymbol{x}$ in the original OCP. Additionally, because the costate $\boldsymbol{p}$ represents the cost sensitivity to state perturbations on the optimal trajectory[109],

$$\boldsymbol{p}^T = \frac{\partial V}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),t) \tag{A.11}$$

which allows the HJB PDE to be rewritten as

$$-\frac{\partial V}{\partial t}(\boldsymbol{x},t) = \underset{\boldsymbol{u}\in U}{\text{opt}}\left[\mathcal{H}(\boldsymbol{x},\boldsymbol{u},\frac{\partial V}{\partial \boldsymbol{x}}^T,t)\right] \tag{A.12}$$

The HJB PDE provides a necessary and sufficient condition for local optimality if the value function is $\mathcal{C}^1$. As a result, a value function $V(\boldsymbol{x},t)$ that solves the HJB PDE describes a hypersurface of optimal cost in the feasible range of $\boldsymbol{x}$ during the solution interval $[t_0,t_f]$ [109]. Since V is defined for all t in the solution interval and x, $V(\boldsymbol{x},t)$ represents the optimal cost-to-go from an arbitrary point $(\boldsymbol{x},t)$ on the trajectory $\boldsymbol{x}$(t). In general, the value function solution is not smooth ($\mathcal{C}^1$). Thus, viscosity solutions of the HJB PDE are often obtained numerically.

To solve the HJB PDE, a boundary condition on the value function is specified. For reachability problems, this generally comes in the form of $V(\boldsymbol{x}_0,t_0)$ whose zero-level set defines the initial reachability set boundary. Using the HJB PDE framework, the reachability set at

any point in time is then defined as

$$\mathcal{R}(t; V(\boldsymbol{x}_0, t_0)) = \{\boldsymbol{x} | V(\boldsymbol{x}, t) \leq 0\} \tag{A.13}$$

where $V(\boldsymbol{x}, t)$ is the solution to the HJB PDE.

A.2 Minimum-time Reachability Optimal Control Problem First Order Necessary Conditions of Optimality

The minimum-time reachability optimal control problem (OCP) has the form of

$$\begin{aligned} \sup_{\boldsymbol{u} \in U} J(\boldsymbol{u}) &= V(\boldsymbol{x}_f, t_f) \\ \text{subject to: } \dot{\boldsymbol{x}} &= \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t) \\ \boldsymbol{g}(\boldsymbol{x}_0, t_0) &= \boldsymbol{0} \end{aligned} \tag{A.14}$$

in which the continuum of solutions to this OCP will define the reachability set at a particular time horizon. At this point in deriving the necessary conditions, the control constraints are ignored. Calculus of variations cannot easily incorporate control constraints but the first order necessary conditions are more simply identified using this approach. Incorporating control constraints will be addressed afterward. After augmenting the initial manifold and dynamics constraints to the total cost using Lagrange multipliers,

$$\sup_{\boldsymbol{u}} \tilde{J}(\boldsymbol{u}) = \int_{t_0}^{t_f} \boldsymbol{p}^T (\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, \tau) - \dot{\boldsymbol{x}}) d\tau + V(\boldsymbol{x}_f, t_f) + \boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x}_0, t_0) \tag{A.15}$$

Introducing the functional variations on $\boldsymbol{u} \rightarrow \boldsymbol{u} + \epsilon \boldsymbol{\nu}$ where ϵ is a small scalar and $\boldsymbol{\nu}$ represents an arbitrary functional variation on the control input trajectory, the resulting variation of the state trajectory is given as $\boldsymbol{x} \rightarrow \boldsymbol{x} + \epsilon \boldsymbol{\eta} + o(\epsilon)$. This result comes from the assumption the dynamics are Lipschitz in $\boldsymbol{x}$ and piece-wise continuous in t. A perturbed

cost can then be expressed as

$$\tilde{J}(\boldsymbol{u}+\epsilon\boldsymbol{\nu}) = \int_{t_0}^{t_f} \boldsymbol{p}^T[\boldsymbol{f}(\boldsymbol{x}+\epsilon\boldsymbol{\eta}, \boldsymbol{u}+\epsilon\boldsymbol{\nu}, \tau)-(\dot{\boldsymbol{x}}+\epsilon\dot{\boldsymbol{\eta}})]d\tau+V(\boldsymbol{x}_f+\epsilon\boldsymbol{\eta}_f, t_f)+\boldsymbol{\lambda}^T\boldsymbol{g}(\boldsymbol{x}_0+\epsilon\boldsymbol{\eta}_0, t_0)+o(\epsilon) \tag{A.16}$$

Expanding each term in a Taylor series expansion results in

$$\begin{aligned}\tilde{J}(\boldsymbol{u}+\epsilon\boldsymbol{\nu}) = \int_{t_0}^{t_f} \boldsymbol{p}^T[\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, \tau) + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\epsilon\boldsymbol{\eta} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\epsilon\boldsymbol{\nu} - (\dot{\boldsymbol{x}}+\epsilon\dot{\boldsymbol{\eta}})]d\tau \\ +V(\boldsymbol{x}_f, t_f) + \frac{\partial V}{\partial \boldsymbol{x}_f}\epsilon\boldsymbol{\eta}_f + \boldsymbol{\lambda}^T(\boldsymbol{g}(\boldsymbol{x}_0, t_0) + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}\epsilon\boldsymbol{\eta}_0) + o(\epsilon)\end{aligned} \tag{A.17}$$

The first order necessary condition of optimality states the first variation of the cost, traditionally expressed as a Gateaux derivative, vanishes along the optimal trajectory

$$\begin{aligned}\delta\tilde{J}(\boldsymbol{u};\boldsymbol{\nu}) &\equiv \lim_{\epsilon\to 0}\frac{\tilde{J}(\boldsymbol{u}+\epsilon\boldsymbol{\nu})-\tilde{J}(\boldsymbol{u})}{\epsilon} = 0 \\ \delta\tilde{J}(\boldsymbol{u};\boldsymbol{\nu}) = \lim_{\epsilon\to 0}\frac{1}{\epsilon}\Bigg[\int_{t_0}^{t_f} \boldsymbol{p}^T[\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\epsilon\boldsymbol{\eta} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\epsilon\boldsymbol{\nu} - \epsilon\dot{\boldsymbol{\eta}}]d\tau + \frac{\partial V}{\partial \boldsymbol{x}_f}\epsilon\boldsymbol{\eta}_f + \boldsymbol{\lambda}^T\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}\epsilon\boldsymbol{\eta}_0) + o(\epsilon)\Bigg] \\ &= \int_{t_0}^{t_f} \boldsymbol{p}^T[\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\boldsymbol{\eta} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\boldsymbol{\nu} - \dot{\boldsymbol{\eta}}]d\tau + \frac{\partial V}{\partial \boldsymbol{x}_f}\boldsymbol{\eta}_f + \boldsymbol{\lambda}^T\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}\boldsymbol{\eta}_0\end{aligned} \tag{A.18}$$

Using integration by parts

$$\int_{t_0}^{t_f} \boldsymbol{p}^T\dot{\boldsymbol{\eta}}d\tau = \boldsymbol{p}_f^T\boldsymbol{\eta}_f - \boldsymbol{p}_0^T\boldsymbol{\eta}_0 - \int_{t_0}^{t_f} \dot{\boldsymbol{p}}^T\boldsymbol{\eta}d\tau \tag{A.19}$$

The Gateaux derivative then reduces to

$$\delta\tilde{J}(\boldsymbol{u};\boldsymbol{\nu}) = \int_{t_0}^{t_f} \big[\boldsymbol{p}^T\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} + \dot{\boldsymbol{p}}^T\big]\boldsymbol{\eta}\, d\tau + \int_{t_0}^{t_f} \big[\boldsymbol{p}^T\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\big]\boldsymbol{\nu}\, d\tau + \big[\frac{\partial V}{\partial \boldsymbol{x}_f} - \boldsymbol{p}_f^T\big]\boldsymbol{\eta}_f + \big[\boldsymbol{\lambda}^T\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0} + \boldsymbol{p}_0^T\big]\boldsymbol{\eta}_0 \tag{A.20}$$

For the first variation of the cost to vanish, each individual term must vanish. Because

the initial or terminal state $(\boldsymbol{x}_0, \boldsymbol{x}_f)$ are not fully specified, the corresponding functional variations $(\boldsymbol{\eta}_0, \boldsymbol{\eta}_f)$ are free. Similarly, there are no control or state constraints along the trajectory so the functional variations $\boldsymbol{\eta}, \boldsymbol{\nu}$ are also free. For each term to vanish for arbitrary variations, the bracketed terms must vanish. As a result, the first order necessary conditions of optimality are given by

$$\begin{aligned} \dot{\boldsymbol{p}} &= -\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}^T (\boldsymbol{x}, \boldsymbol{u}, t)\, \boldsymbol{p} \\ \boldsymbol{p}_f &= \frac{\partial V}{\partial \boldsymbol{x}_f}^T (\boldsymbol{x}_f, t_f) \\ \boldsymbol{p}_0 &= -\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}^T (\boldsymbol{x}_0, t_0)\boldsymbol{\lambda} \\ \boldsymbol{g}(\boldsymbol{x}_0, t_0) &= \mathbf{0} \end{aligned} \tag{A.21}$$

along with the optimal control $\boldsymbol{u}^*$ defined by

$$\boldsymbol{p}^T \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}^*} = \mathbf{0} \tag{A.22}$$

where $\boldsymbol{p}$ is the costate variable for the state $\boldsymbol{x}$ and $\boldsymbol{\lambda}$ is the Lagrange multiplier for the state boundary constraint $\boldsymbol{g}(\boldsymbol{x}_0, t_0) = \mathbf{0}$. To solve the OCP given in Eq. (A.14), trajectories for the state and costate must be found to simultaneously satisfy Eq. (A.21) under the optimal control policy defined by Eq. (A.22). Because the state dynamics are assumed to be at least Lipschitz continuous, the state and costate trajectories are uniquely determined by their boundary condition groups $(\boldsymbol{x}_0, \boldsymbol{p}_0, t_0)$ or $(\boldsymbol{x}_f, \boldsymbol{p}_f, t_f)$. It is convenient to define the trajectories of the state and costate using trajectory or flow functions as follows

$$\begin{aligned} \boldsymbol{x}(t) &= \phi_x(t; \boldsymbol{x}_0, \boldsymbol{p}_0, t_0) \\ \boldsymbol{p}(t) &= \phi_p(t; \boldsymbol{x}_0, \boldsymbol{p}_0, t_0) \end{aligned} \tag{A.23}$$

This notation denotes that given an initial condition group, the resulting trajectory can be

evaluated at any time t. Rewriting the transversality conditions in Eq. (A.21) in terms of these flow functions yields

$$\begin{aligned}\boldsymbol{p}_f = \phi_p(t_f; \boldsymbol{x}_0, -\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}^T(\boldsymbol{x}_0, t_0)\boldsymbol{\lambda}, t_0) = \frac{\partial V}{\partial \boldsymbol{x}_f}^T(\phi_x(t_f; \boldsymbol{x}_0, -\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}^T(\boldsymbol{x}_0, t_0)\boldsymbol{\lambda}, t_0), t_f) \\ \boldsymbol{g}(\phi_x(t_0; \boldsymbol{x}_0, -\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}_0}^T(\boldsymbol{x}_0, t_0), t_0)\boldsymbol{\lambda}, t_0) = \mathbf{0}\end{aligned} \tag{A.24}$$

This reduces the solution to the OCP to finding $(\boldsymbol{x}_0, \boldsymbol{\lambda})$ that simultaneously satisfy Eq. (A.24) under the optimal control policy defined by Eq. (A.22).

With the traditionally defined Hamiltonian in this problem,

$$\mathcal{H}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t) = \boldsymbol{p}^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t) \tag{A.25}$$

the state and costate dynamics can be rewritten as

$$\dot{\boldsymbol{p}} = -\frac{\partial \mathcal{H}}{\partial \boldsymbol{x}}^T \tag{A.26a}$$

$$\dot{\boldsymbol{x}} = \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}}^T \tag{A.26b}$$

along with the optimal control $\boldsymbol{u}^*$ defined by

$$\frac{\partial \mathcal{H}}{\partial \boldsymbol{u}^*} = \mathbf{0} \tag{A.27}$$

Eq. (A.26) reveals the state and costate form a Hamiltonian system characteristic of the traditional definition in dynamical systems theory. For the OCP with unconstrained control input, the solution to Eq. (A.21) and Eq. (A.22) give the optimal trajectories for the state, costate, and control. However, control input constraints are common and cannot be completely addressed with Eq. (A.22). To address control constraints, the Pontryagin Max-

imum Principle in Eq. (A.28) is used to generalize the optimal control policy expressed in Eq. (A.27)

$$\boldsymbol{u}^* = \underset{\boldsymbol{u}\in U}{\text{argmax}} \{\mathcal{H}\} \tag{A.28}$$

which generalizes the generation of the optimal control policy along a trajectory to pointwise optimizations of the Hamiltonian.

A.3 Minimum-Time Optimal Control Policy for Control Affine Systems

Section 2.2.3 discusses the use of feasible control sets that are defined using affine transformations of unit balls of the p-norm and F-norm vector spaces. This allows for non-ellipsoidal feasible control sets in a given reachability analysis. Moreover, the feasible control set does not need to be centered around the origin or contain the origin at all. This allows for minimum-time reachability analyses in which feasible control set defines the allowable deviations from a nominal control input signal, $\mathbf{u}_c(t)$ or $\mathbf{u}_c(\mathbf{x}, t)$.

From Section 2.2.4, the following optimization problem is presented.

$$\begin{aligned} &\max_{\mathbf{x}} \mathbf{y}^T\mathbf{x} \\ &s.t.\ \|\mathbf{x}\|_p \leq 1 \end{aligned} \tag{A.29}$$

which has the following analytic solution

$$\mathbf{x}^* = \frac{\text{sign}(\mathbf{y}) \circ |\mathbf{y}|^{q-1}}{\|\mathbf{y}\|_q^{q-1}} = \frac{\partial \|\mathbf{y}\|_q}{\partial \mathbf{y}} \tag{A.30}$$

where $q = \frac{p}{p-1}$, $\circ$ denotes the Hadamard/element-wise product, $|\cdot|$ denotes the element-wise absolute value, and $\text{sign}(\cdot)$ denotes the element-wise sign/signum operation.

As shown in Section 2.2.2, normed unit balls, $\|\tilde{\mathbf{x}}\| \leq 1$, can be defined using the affine

transformation $\tilde{\mathbf{x}} = M(\mathbf{x} - \mathbf{x}_c)$, where $\mathbf{x}_c$ defines the center of the feasible set and $M \in \mathbb{R}^{n \times n}$ is an invertible transformation matrix. An equivalent definition of the feasible set is given by $X = \{\mathbf{x} \in \mathbb{R}^n \ : \ \mathbf{x} = M^{-1}\tilde{\mathbf{x}} + \mathbf{x}_c, \|\tilde{\mathbf{x}}\| \leq 1\}$.

For the minimum-time optimal control problem, Pontryagin's maximum principle states the optimal control input comes in the form

$$\mathbf{u}^* = \underset{\mathbf{u} \in U}{\operatorname{argmax}} \{\mathcal{H}\} = \underset{\mathbf{u} \in U}{\operatorname{argmax}} \{\mathbf{p}^T \mathbf{f}(\mathbf{x}, \mathbf{u}, t)\} \tag{A.31}$$

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. One prominent example of this is given by control affine dynamic systems.

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

$$\dot{\mathbf{x}} = \mathbf{f}_1(\mathbf{x}, t) + \mathbf{f}_2(\mathbf{x}, t)\mathbf{u} \tag{A.32}$$

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

$$\begin{aligned} \mathbf{u} &= M^{-1}\tilde{\mathbf{u}} + \mathbf{u}_c \\ \|\tilde{\mathbf{u}}\|_p &\leq 1 \end{aligned} \tag{A.33}$$

as described above, the Pontryagin's maximum principle comes in the form

$$\begin{aligned}
\mathbf{u}^* &= \underset{\mathbf{u}\in U}{\text{argmax}}\ \mathbf{p}^T\left[\mathbf{f}_1(\mathbf{x},t) + \mathbf{f}_2(\mathbf{x},t)\mathbf{u}\right] \\
&= \underset{\mathbf{u}\in U}{\text{argmax}}\ \mathbf{p}^T\mathbf{f}_2(\mathbf{x},t)\mathbf{u} \\
&= \underset{\|\tilde{\mathbf{u}}\|_p\leq 1}{\text{argmax}}\ \mathbf{p}^T\mathbf{f}_2(\mathbf{x},t)\left[M^{-1}\tilde{\mathbf{u}} + \mathbf{u}_c\right] \\
&= \underset{\|\tilde{\mathbf{u}}\|_p\leq 1}{\text{argmax}}\ \mathbf{p}^T\mathbf{f}_2(\mathbf{x},t)M^{-1}\tilde{\mathbf{u}}
\end{aligned} \tag{A.34}$$

The maximum principle in Eq. (A.34) is equivalent to the optimization problem in Eq. (A.29) where $\mathbf{y} = M^{-T}\mathbf{f}_2^T(\mathbf{x},t))\mathbf{p}$. Consequently, the analytic solution for the optimal control policy for control affine systems is given by

$$\begin{aligned}
\tilde{\mathbf{u}}^* &= \frac{\text{sign}(\mathbf{y})\circ|\mathbf{y}|^{q-1}}{\|\mathbf{y}\|_q^{q-1}} = \frac{\partial\|\mathbf{y}\|_q}{\partial\mathbf{y}} \\
\mathbf{u}^* &= M^{-1}\tilde{\mathbf{u}}^* + \mathbf{u}_c
\end{aligned} \tag{A.35}$$

where $\mathbf{y} = M^{-T}\mathbf{f}_2^T(\mathbf{x},t))\mathbf{p}$ and $q = \frac{p}{p-1}$.

Note that this form has a well-defined solution for $1 < p < \infty$. This corresponds to having a smooth, continuously differentiable boundary for the feasible set. Outside of this range, solutions exist but may not be unique in terms of $\mathbf{y}$.

For the case of $p = 1$,

$$\begin{aligned}
|\tilde{u}_i^*| &= \begin{cases} \text{sign}(y_i) & \textit{if } i = a \\ 0 & \text{otherwise} \end{cases} \\
\mathbf{u}^* &= M^{-1}\tilde{\mathbf{u}}^* + \mathbf{u}_c
\end{aligned} \tag{A.36}$$

where a is the index of $\mathbf{y}$ where $|y_i|$ is the largest.

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

$$\begin{aligned} \tilde{\mathbf{u}}^* &= \text{sign}(\mathbf{y}) \\ \mathbf{u}^* &= M^{-1}\tilde{\mathbf{u}}^* + \mathbf{u}_c \end{aligned} \tag{A.37}$$

For the special case of $p = 2$ corresponding to spherical/circular feasible sets,

$$\begin{aligned} \tilde{\mathbf{u}}^* &= \frac{\mathbf{y}}{\|\mathbf{y}\|_2} \\ \mathbf{u}^* &= M^{-1}\tilde{\mathbf{u}}^* + \mathbf{u}_c \end{aligned} \tag{A.38}$$

APPENDIX B

REPRODUCING RESULTS

B.1 Viscous Damper Linear System

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

B.1.1 Problem Setup

$$\ddot{x} = -\mu\dot{x} + u, \dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ 0 & -\mu \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{B.1}$$

With control input constraints given by

$$|u| \leq 1 \tag{B.2}$$

and initial condition constraint given by

$$\begin{aligned} g(\mathbf{x}_0, t_0) &= \|M\mathbf{x}\|_8 - 1 \leq 0 \\ M &= \begin{bmatrix} \sqrt{\frac{1}{2}} & 1 \\ 0 & \sqrt{\frac{1}{10}} \end{bmatrix} \end{aligned} \tag{B.3}$$

B.1.2 Imports

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

B.1.3 Create Dynamic System Object

```
        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)
```

B.1.4 Create Initial Condition Constraint

```
        E = np.diag([0.5,0.1])
        IC = Constraint.fromShapeMatrix(E,p=8)
```

B.1.5 Create Reachability Object and Initialize Particles

```
        R = Reachability(LTI,IC)
        R.initializeReach(subspaceDim=2,␣
↪uniformSamplingRate=60)
```

B.1.6 Compute Forward Reachable Set over Time Horizon

```
        timeSteps = 50
        t0 = 0.0
        tf = 2.0
        Tvec = np.linspace(t0, tf, timeSteps)
        R_FRS.computeReach(Tvec)
```

B.1.7 Compute Trajectories

```
        R_FRS.
↪computeCurrentStateTrajectories(numTimeSteps=timeSteps)
        R_FRS.computeFinalStateHistories()
```

B.1.8 Convert to Forward Reachable Tube

```
        R_FRT = R_FRS.convertReachSetToTube(returnNew=True)
```

B.1.9 Compute Backward Reachable Set over Time Horizon

```
        timeSteps = 50
        t0 = 0.0
        tf = -2.0 # equivalent to t0 = 2.0, tf = 0.0
        Tvec = np.linspace(t0, tf, timeSteps)
        R_BRS.computeReach(Tvec)
```

B.1.10 Compute Trajectories

```
        R_BRS.
↪computeCurrentStateTrajectories(numTimeSteps=timeSteps)
        R_BRS.computeFinalStateHistories()
```

B.1.11 Convert to Backward Reachable Tube

```
        R_BRT = R_BRS.convertReachSetToTube(returnNew=True)
```

B.1.12 Plot results

```
        fh1 = plotReachGrowth(R_FRS.particles)
        fh2 = plotReachGrowth(R_FRT.particles)
        fh3 = plotReachGrowth(R_BRS.particles)
        fh4 = plotReachGrowth(R_BRT.particles)
```

B.2 Zermelos 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 Zermelos problem.

B.2.1 Problem Setup

$$\begin{aligned} \dot{x} &= V_m \cos(\theta) + w_x(x, y) \\ \dot{y} &= V_m \sin(\theta) + w_y(x, y) \\ V_m &= 1 \\ w_x &= 1 \\ w_y &= x^2 \end{aligned} \tag{B.4}$$

With control input constraints given by

$$\theta \in [0, 2\pi) \tag{B.5}$$

and initial condition constraints are given by

$$\begin{aligned} g_1(\mathbf{x}_0) &= 9(x_0 + 1)^2 + 100y_0^2 \leq 1 \\ g_2(\mathbf{x}_0) &= 100(x_0 + 1)^2 + 9y_0^2 \leq 1 \end{aligned} \tag{B.6}$$

B.2.2 Create Nonlinear Dynamics Model Using Python Symbolic Toolbox

Imports

```
        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.

B.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)
```

(continues on next page)

(continued from previous page)

```
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)
```

B.3 Duffing Oscillator - Mesh Refinement

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

B.3.1 Problem Setup

$$\begin{aligned} m_1\ddot{x}_1 &= -k_{1,1}x_1 - k_{1,3}x_1^3 - f_1\dot{x}_1 \\ +k_{2,1}(x_2 - x_1) + k_{2,3}(x_2 - x_1)^3 &+ f_2(\dot{x}_2 - \dot{x}_1) \end{aligned} \tag{B.7}$$

$$
\begin{aligned}
m_2\ddot{x}_2 = \\
-k_{2,1}(x_2 - x_1) - k_{2,3}(x_2 - x_1)^3 - f_2(\dot{x}_2 - \dot{x}_1) \\
+k_{3,1}(x_3 - x_2) + k_{3,3}(x_3 - x_2)^3 + f_3(\dot{x}_3 - \dot{x}_2)
\end{aligned} \tag{B.8}
$$

$$
\begin{aligned}
m_3\ddot{x}_3 = -k_{3,1}(x_3 - x_2) - k_{3,3}(x_3 - x_2)^3 \\
-f_3(\dot{x}_3 - \dot{x}_2) + u
\end{aligned} \tag{B.9}
$$

$$
\begin{aligned}
m_1 = m_2 = m_3 = 1 \quad k_{1,1} = k_{2,1} = k_{3,1} = 1 \\
k_{1,3} = k_{2,3} = k_{3,3} = 1/9 \quad f_1 = f_2 = f_3 = 1 \\
u_m = 1, T_f = \pi
\end{aligned} \tag{B.10}
$$

With control input constraints given by

$$
|u| \leq 1 \tag{B.11}
$$

and initial condition constraints are given by

$$
g(\mathbf{x}_0) = \mathbf{x}_0^T\mathbf{x}_0 - 1 = 0 \tag{B.12}
$$

B.3.2 Create Nonlinear Dynamics Model Using Python Symbolic Toolbox

Imports

```
        from sympy import symbols, Matrix, simplify, eye,␣
↪zeros, flatten
        from sympy.utilities.autowrap import autowrap
        from utilities import cpickleSave
```

(continues on next page)

(continued from previous page)

```
        from utilities import sympyNorm, sympyAbsApprox,␣
↪sympySignApprox
```

Define Variables, State, and Costate

```
        # 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, p1d ,p2d, p3d) = symbols('p1, p2, p3,␣
↪p1d ,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])
        # Define costate vector
        p = Matrix([p3,p3d,p1,p2,p1d,p2d])
        # 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)
        uLimit = [1.] # max possible input in each control
↪dimension
        Minv = eye(m) # initialize inv(M) matrix
        for i in range(m):
        Minv[i,i] = uLimit[i] # place uLimit along diagonal
        B = Matrix([[0],[1],[0],[0],[0],[0]])
        y = Minv.T*B.T*p
        pNorm = 2.0 # for second option, pNorm is always
↪equal to 2
        s = 0.0
        s2 = sympyNorm(y)**2.0
        for i in range(m):
        s += sympyAbsApprox(y[i],scalar=True)**(pNorm/
↪(pNorm-1.0))
        uTilde = zeros(m,1)
        for i in range(m):
        uTilde[i] = sympySignApprox(y[i],
↪scalar=True)*sympyAbsApprox(y[i],scalar=True)**(1.0/
↪(pNorm-1.0))/(s**(1.0/pNorm))
        u = ts*simplify(Minv*uTilde) # smooth approximation
↪of uStar/optimal control (no need for switch)
        u2 = ts*simplify(Minv*y/sympyNorm(y)) # analytic
↪uStar for p=2 only (need for switch when s2==0)
```

(continues on next page)

(continued from previous page)

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

# Define state dynamics
x21 = x2 - x1
x32 = x3 - x2
x21d = x2d - x1d
x32d = x3d - x2d

x1dd = 1/m1*(-k11*x1 - k13*x1**3.0 - f1*x1d
+ k21*x21 + k23*x21**3.0 + f2*x21d)
x2dd = 1/m2*(-k21*x21 - k23*x21**3.0 - f2*x21d
+k31*x32 + k33*x32**3.0 + f3*x32d)
x3dd = 1/m3*(-k31*x32 - k33*x32**3.0 - f3*x32d)
xd = Matrix([x3d,x3dd,x1d,x2d,x1dd,x2dd])
f = xd + B*u
f2 = xd + B*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.

B.3.3 Perform Reachability Analysis

Imports

```
        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

```
        Sym1 = SympyDynamicsSmooth(n=6,fname='duffing_x3_
↪option1.modname')
        Sym2 = SympyDynamicsSwitch(n=6,fname='duffing_x3_
↪option2.modname')
```

Create Initial Condition Constraints

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

Create Reachability Objects and Initialize Particles

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

Compute Forward Reachable Sets over Time Horizon

```
        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

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

(continues on next page)

(continued from previous page)

```
        R2_cubic.meshRefinement(method='cubic',cost='J',
↪q1=25,q3=65,maxRefinements=3)
```

Plot results

```
        # Plot results!
        plotReachTrajectories(R2.particles,projDim=[1,2],
↪addedPtInds=list(range(R2.origNumParticles,R2.
↪numParticles)))
        plotReachVolume(R2.particles,projDim=[1,2])
        plotRedistributionCDFs(R2.meshRefinementBeforeAfter,
↪idealPlotBool=True,columnBool=False)


        plotReachTrajectories(R2_cubic.particles,projDim=[1,
↪2],addedPtInds=list(range(R2_cubic.origNumParticles,R2_
↪cubic.numParticles)))
        plotReachVolume(R2_cubic.particles,projDim=[1,2])
        plotRedistributionCDFs(R2_cubic.
↪meshRefinementBeforeAfter,idealPlotBool=True,
↪columnBool=False)
```

B.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.

B.4.1 Problem Setup

$$
\begin{aligned}
\mathbf{X} &= [x \;\; y \;\; \mathring{x} \;\; \mathring{y} \;\; J_1]^T \\
\mathring{\mathring{x}} &= 2\mathring{y} + x - (1-\mu)\frac{x-x_1}{\rho_1^3} - \mu\frac{x-x_2}{\rho_2^3} + \bar{u}_x \\
\mathring{\mathring{y}} &= -2\mathring{x} + \left(1 - \frac{1-\mu}{\rho_1^3} - \frac{\mu}{\rho_2^3}\right) y + \bar{u}_y \\
J_1 &= -\int_0^{\tau_f} \|\bar{\mathbf{u}}\|_1 d\tau = -\int_0^{\tau_f} |\bar{u}_x| + |\bar{u}_y| d\tau
\end{aligned}
\tag{B.13}
$$

With control input constraints given by

$$
\begin{aligned}
|u_x| &\leq 5.714e-5 \text{ m/s2} \\
|u_y| &\leq 5.714e-5 \text{ m/s2}
\end{aligned}
\tag{B.14}
$$

and initial condition constraints are given by

$$
\begin{aligned}
g(\mathbf{X}_0) = \left(\frac{x-x_{L1}}{\epsilon_1}\right)^2 + \left(\frac{y}{\epsilon_1}\right)^2 + \left(\frac{\mathring{x}}{\epsilon_2}\right)^2 + \left(\frac{\mathring{y}}{\epsilon_2}\right)^2 + \left(\frac{J_1}{\epsilon_2}\right)^2 = 0 \\
\epsilon_1 = 1e-3 \\
\epsilon_2 = 1e-4
\end{aligned}
\tag{B.15}
$$

B.4.2 Create Nonlinear Dynamics Model Using Python Symbolic Toolbox

Imports

```
        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
```

(continues on next page)

(continued from previous page)

Define Variables, State, and Costate

```
        # 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)
```

(continues on next page)

(continued from previous page)

```
        # Define problem parameters
        Tmax = (0.8e-3)/14 # m/s2, max thrust [Lunar␣
↪Icecube from Bosanac paper, 14kg and 0.8mN thrust]
        r12 = 384402e3 # m, distance between m1 and m2
        G = 6.67430e-11 # m3/kg/s2, 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 optimal control policy (if it's␣
↪analytically possible)
        uLimit = [Tmax,Tmax] # max possible input in each␣
↪control dimension
        u = zeros(m,1)
        u2 = zeros(m,1)
        B = Matrix([[0,0],[0,0],[0,0],[1,0],[0,1]])
        y = B.T*p
        u_L1_sum = 0
        for i in range(m):
        y2 = y[i]
        u2[i] = uLimit[i]/2*(sympySignApprox(y2 - pJ1,
↪epsil=1.0e-5,scalar=True) + sympySignApprox(y2 + pJ1,
↪epsil=1.0e-5,scalar=True))
        u_L1_sum += sympyAbsApprox(u2[i],scalar=True)
```

Define State Dynamics

```
        x1dd = 2*x2d + x1 - (1-mu)*(x1-x1_)/(rho1**3) -␣
↪mu*(x1-x2_)/(rho2**3)
        x2dd = -2*x1d + (1 - (1-mu)/(rho1**3) -  - mu/
↪(rho2**3) )*x2


        Jd_L1 = Matrix([-u_L1_sum])
```

(continues on next page)

(continued from previous page)

```
        xd_L1 = Jd_L1.col_join(Matrix([x1d,x2d,x1dd,x2dd]))
        f = xd_L1 + B*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.

B.4.3 Perform Reachability Analysis

Imports

```
        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

```
        Sym2 = SympyDynamicsSmooth(n=5,fname='cislunar2_
↪option2.modname')
```

Create Initial Condition Constraints

```
        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

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

Compute Forward Reachable Sets over Time Horizon

```
        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 = 2.6653296644361014e-06 # angular velocity of EM
↪system in rad/s
        r12 = 384402000.0 # distance from Earth to moon in
↪meters
        r12_km = r12/1000 # distance from Earth to moon in
↪kilometers
        nonDim2SI2 = w*r12 # converts J_1 to deltaV in m/s


        # Plot results!
        fh1 = plotReachTrajectories(R.particles,projDim=[2,
↪3,1],axisEqualBool=False,sc=[r12_km,r12_km,nonDim2SI2],
↪axesLabels=['$x$ [km]','$y$ [km]','$J_1$ [m/s]'])
        fh2 = plotReachTrajectories(R.particles,projDim=[3,
↪1],axisEqualBool=False,sc=[r12_km,nonDim2SI2],axesLabels=[
↪'$y$ [km]','$J_1$ [m/s]'])
        fh3 = plotReachTrajectories(R.particles,projDim=[2,
↪1],axisEqualBool=False,sc=[r12_km,nonDim2SI2],axesLabels=[
↪'$x$ [km]','$J_1$ [m/s]'])
        fh4 = plotReachVolume(R.particles,projDim=[2,3,1],
↪axisEqualBool=False,sc=[r12_km,r12_km,nonDim2SI2],
↪axesLabels=['$x$ [km]','$y$ [km]','$J_1$ [m/s]'])
        fh5 = plotReachVolume(R.particles,projDim=[2,3],
↪axisEqualBool=False,sc=[r12_km,r12_km],axesLabels=['$x$
↪[km]','$y$ [km]'])
```

(continues on next page)

(continued from previous page)

```
        fh6 = plotReachHistories1D(R.particles,projDim=1,
↪shadedTrajBool=True,particleHistBool=True,sc=nonDim2SI2,
↪axesLabels=['T','$J_1$ [m/s]'])
```

Save/Export Results

```
        vArr1, pArr1 = R.vertices('xf_T',True)
        vArr2, pArr2 = R.vertices('x',True)
        vArr3, pArr3 = R.vertices_over_T(True)
        from scipy.io import savemat

        savemat('cislunarWayMoreParticles5_over_T_Again.mat
↪', {
        'vArr1': vArr1,
        'pArr1': pArr1,
        'vArr2': vArr2,
        'pArr2': pArr2,
        'vArr3': vArr3,
        'pArr3': pArr3,
        'Tvec': R.Tvec,
        })
```

APPENDIX C

SCORE DOCUMENTATION

C.1 Reachability Toolbox Tutorial

C.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

C.1.2 Example Scenario

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

$$\ddot{x} = u, \dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u, |u| \leq 1 \tag{C.1}$$

With control input constraints given by

$$|u| \leq 1 \tag{C.2}$$

and initial condition constraint given by

$$V(\mathbf{x}_0, t_0) = \begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix}^T \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix} - 1 \leq 0 \tag{C.3}$$

Imports

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

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

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:

```
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.

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:

```
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.

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:

```
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:

```
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.

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:

```
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.

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:

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

Plot results

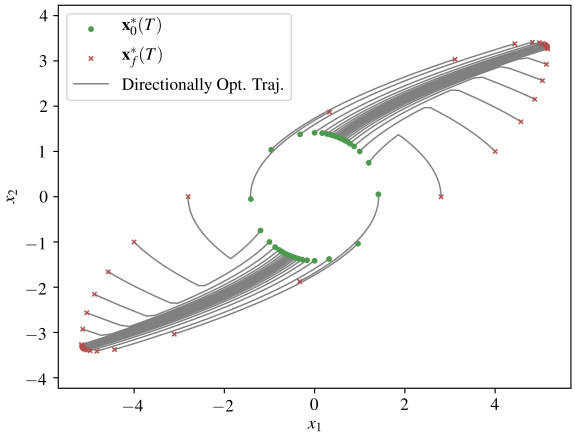
There are a large number of visualizing 2D and 3D reachability results provided. Examples of a few are shown here:

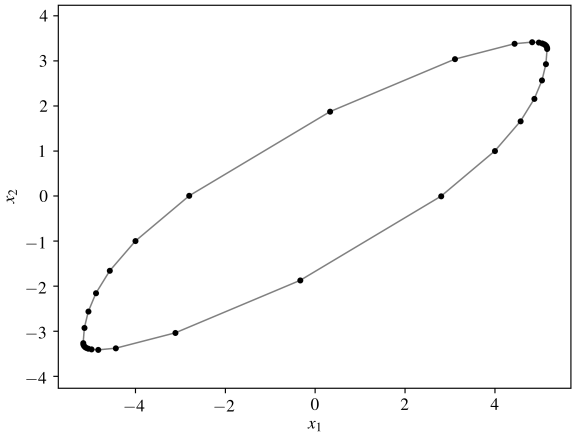
```
fh1 = plotReachTrajectories(R.particles)
fh2 = plotReach(R.particles)
fh3 = plotReachGrowth(R.particles)
fh4 = plotReachTrajectories1D(R.particles, projDim=1,␣
↪shadedTrajBool=True, particleTrajBool=True)
```

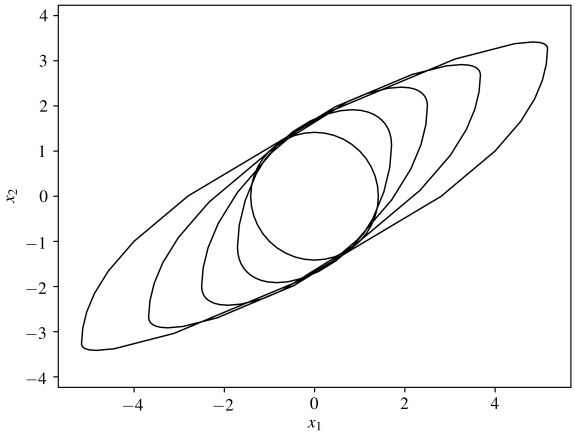
(continues on next page)

(continued from previous page)

```
fh5 = plotReachTrajectories1D(R.particles, projDim=2,␣
↪shadedTrajBool=True, particleTrajBool=True)
```







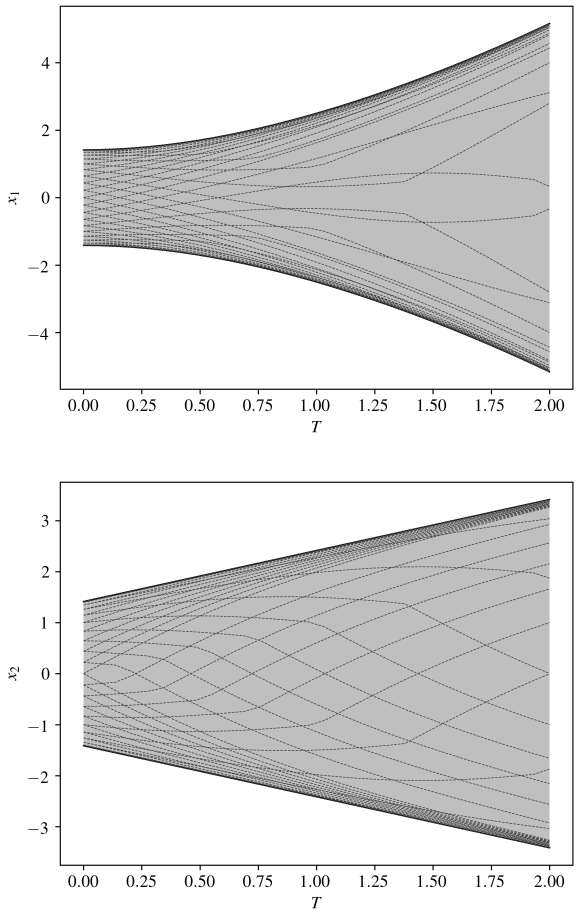


Figure C.1: Example figures from forward reach set analysis

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:

```
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:

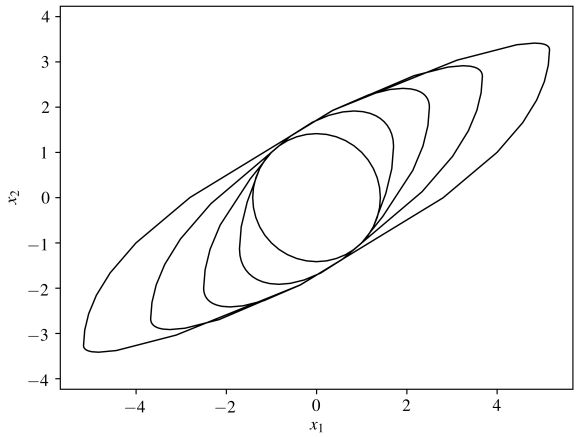


Figure C.2: Forward reachable tube converted from reachable set

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

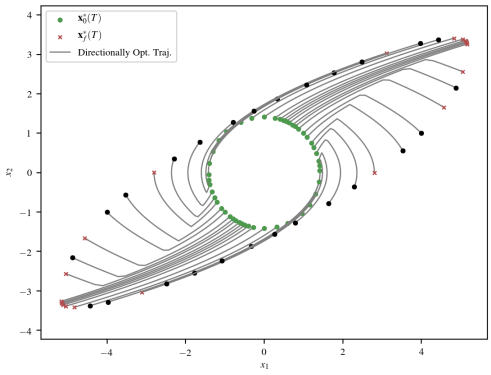


Figure C.3: Additional particles added through bisection mesh refinement

Save Data

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

using:

```
R.saveFigures([fh1, fh2, fh3, fh4, fh5, fh7], saveFilename=
↪'doubleIntegrator_FRS')
R2.saveFigures([fh6], saveFilename='doubleIntegrator_FRT')
```

The easiest way to save the entire reachability analysis is to use the dill module:

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

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

```
import dill
dill.load_session('doubleIntegrator_session.pkl')
```

C.2 Notation

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.

C.2.1 Reachability terms

Table C.1: Reachability Notation Table

Name	Description
Reachable Set or RS	Volume of state space representing all achievable states exactly at the given time horizon T originating from the initial condition volume
Reachable Tube or RT	Volume of state space representing all achievable states up to the given time horizon T originating from the initial condition volume
Forward Reachable Volume	Reachable volume of state space where t0 $<$ tf. In this case, the reachable volume must originate from the initial condition set and change as t goes from t0 to tf
Backwards Reachable Volume	Reachable volume of state space where tf $<$ t0. In this case, the reachable volume must terminate at the initial condition set and change as t goes from tf to t0
Subspace Reachable Volume	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.
Subspace of interest	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.
subspaceDim	Number of dimensions in the subspace of state space to perform the reachability analysis
Performance Metric or V	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
Particle or Sample	A sample of the reachable volume boundary. Each particle corresponds to the solution to a single optimal control problem

C.2.2 Vector terms

Table C.2: Vector Notation Table

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

C.2.3 Time notation

Table C.3: Time Notation Table

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

C.3 Optimal Control Overview

Optimal control review is listed here

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

$$\max_{\mathbf{u}\in U} V(\mathbf{x}_f, t_f) s.t. \; \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \mathbf{g}(\mathbf{x}_0, t_0, \mathbf{x}_f, t_f) = \mathbf{0} \tag{C.4}$$

where $V(\mathbf{x}_f, t_f)$ denotes the performance metric or objective function for the optimal control problem, U denotes the control input constraint, and $\mathbf{g}(\mathbf{x}_0, t_0)$ denotes the initial condition constraint.

The optimal control input can be computed using Pontryagins maximum principle as

$$\mathbf{u}^* = \underset{\mathbf{u}\in U}{\operatorname{argmax}}\ \mathbf{p}^T\mathbf{f}(\mathbf{x},\mathbf{u},t) \tag{C.5}$$

where $\mathbf{p}$ 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.

C.3.1 Control Affine Dynamics

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

$$\dot{\mathbf{x}} = \mathbf{f}_1(\mathbf{x},t) + \mathbf{f}_2(\mathbf{x},t)\mathbf{u} \tag{C.6}$$

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

$$\mathbf{u} = M^{-1}\tilde{\mathbf{u}} \| \tilde{\mathbf{u}} \|_p \leq 1 \tag{C.7}$$

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

$$\mathbf{c} = M^{-T}\mathbf{f}_2^T(\mathbf{x},t))\mathbf{p}s = \sum_i |\mathbf{c}_i|^{\frac{p}{p-1}}\tilde{\mathbf{u}}_i^* = \frac{\operatorname{sign}(\mathbf{c}_i)|\mathbf{c}_i|^{\frac{1}{p-1}}}{s^{\frac{1}{p}}}\mathbf{u}^* = M^{-1}\tilde{\mathbf{u}}^* \tag{C.8}$$

For the special case of $p = 2$,

$$\tilde{\mathbf{u}}^* = \frac{\mathbf{c}}{\| \mathbf{c} \|_2}\mathbf{u}^* = M^{-1}\tilde{\mathbf{u}}^* \tag{C.9}$$

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

$$\tilde{\mathbf{u}}^* = \text{sign}(\mathbf{c})\mathbf{u}^* = M^{-1}\tilde{\mathbf{u}}^* \tag{C.10}$$

C.4 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 a 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 ith 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 per-

formance 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.

C.5 Particle Class

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 *ds* 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 $t = t0$ 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 its 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:

- *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*
- *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 conditon 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*
- *xf_T*, *pf_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*

C.6 Constraint Class

This class represents p-norm constraints and provides functions for evaluating useful quanities 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***, $\tilde{x} = Mx$ 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***, $x^T Ex = x^T M^T Mx \leq 1$ 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)

C.7 Dynamic System Classes

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:

$\dot{x} = A(v) * (x - x_v(v)) + B(v)(u - u_v(v)) - h(e_v(v), x_v(v)) + H(x)$

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 quanitites 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)
- propagate_y(y0,t0,tf,numTimeSteps,rtol,atol)
- propagate_Y(y0,t0,tf,numTimeSteps,rtol,atol)

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*, *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.

C.8 Module Code Documentation

C.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 π) 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 Sards 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 newtons 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 = t0T)

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 *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 *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 numTimeSteps 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()

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 particles

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()

Updates the value of the graph distance cost quantities (J) based on current (at current T,ds/th) particle states

updateDistanceCost()

Updates the value of the distance cost quantities (D) based on current (at current T,ds/th) particle states

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 Newtons 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 Newtons 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

C.8.2 Particle Module

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 conditon 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_t_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)

__**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,lam) 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 xf,pf,yf,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 xf,pf,yf,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 particles support function using this particles unit vector search direction (ds).

If not provided, uses *self.x* from particle to compute its 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 particles performance value using this particles unit vector search direction (ds).

If not provided, uses *self.x* from particle to compute its 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 particles performance value using this particles unit vector search direction (ds).

If not provided, uses *self.x* from particle to compute its 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 particles optimality constraint function

Returns **FVal** – Optimality constraint value

Return type array_like, shape (N+1,)

FNorm()

Evaluate 2-norm of particles 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

C.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 quanities 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. 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*, $x^T Ex = x^T M^T Mx \leq 1$ 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)

__**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 $\|\tilde{x}\|_{p,F} = 1, M > 0$, 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

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

classmethod fromShapeMatrix(*E*, *p=2*, *xc=None*, *constType='F'*)

Alternate constructor for constraint class using shape matrix transpose(M)*M

Shape matrix based from ellipsoid equation, $x^T Ex = x^T M^T Mx \leq 1$

Parameters

- **E** (*array_like, shape (N,N)*) – Shape matrix where $x^T Ex = x^T M^T Mx \leq 1$
- **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
- **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 its 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** ({'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

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** ({*'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 **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 doesnt exist (its 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

- **x** (*array_like, shape (N,)*) – State to evaluate the constraint hessian
- **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 con-

straint type (self.constType)

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

Returns **D2gVal** – Constraint hessian matrix at given state, x

Return type array_like, shape (N,N)

D3g(*x*, *constType=None*, *eps=1e-06*)

Evaluate constraint triple derivative at specified state location. Equivalent to d3(||x||_{p,F})/dx3

If p >= 3 or p = 2, the exact solution will be computed and returned. However, for $1 < p < 2$ and $2 > p > 3$, this derivative doesnt exist (its infinite). In these cases, 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

- **x** (*array_like, shape (N,)*) – State to evaluate the constraint triple derivative
- **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)
- **eps** (*float, optional*) – Scalar determining the degree

of sharpness in the smooth approximations. As *eps* approaches 0, the approximations approach the true functions when component of x is near zero

Returns **D3gVal** – Constraint hessian matrix at given state, x

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

randomSample(*scaleFactor=None*)

Computes a random sample that lies on or within the constraint

Parameters **scaleFactor**(*float, 0 < scaleFactor <= 1, optional*) – If scaleFactor isnt 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,)

maxInnerProduct(*y*, *returnLambda=False*, *constType=None*)

Returns the point x that maximizes inner product between y and x (dot(y,x)) where x satisifies the constraint.

Parameters

- **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 (dot(y,x)) where x satisifies 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 **dxdy** – 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 **dxdydy** – 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 **dxdy** – Jacobian of constraint maximal inner product function with respect to the input vector

Return type array_like, shape (N,N)

C.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, 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

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(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,)

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.**LTVDynamics**(*A*, *B*, *uLimit=1*, *pNorm=2*, *t0=0.0*, *signApproxEps=1e-06*)

Bases: object

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

xDot = A(t)*x + B(t)*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*, *t0=0.0*, *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, 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

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(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,)

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.**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(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(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

- **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(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.**dynSystemTemplate**

Bases: `object`

Template for dynamic system class

__init__()

Constructor for Dynamic System Class

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(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(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)

C.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*, *dh_detFunc=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_dx2Func : 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

dPhidt (*Y*, *t*)

STM Dynamics

dYdt (*Y*, *t*)

Augmented Flow dynamics

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

Propagate y0 = x0,p0 from t0 to yf = xf,pf at tf

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

Propagate Y0 = x0,p0,Phi0 from t0 to yf = xf,pf,Phif at tf

plotqLPVFit (*new_v=None*, *showAxes=False*)

Plots qLPV spline fit with all of the table lookup quantities

C.8.6 Continuation Module

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 (numTimeHorizonSteps,)*) – Array of time horizon, T, values to evaluate the continuation method over
- **particle** (*Particle object*) – Current particle in reachability analysis. This is parametrized by a hyperspherical coordinate or unit vector search direction
- **dynamics** (*Dynamics object*) – Dynamics system object that can propagate the state and costate to the specified time horizon
- **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 Sards theorem (NOT WORKING)

4. Predictor-Corrector with Newton step updates (IN DEVELOPMENT)

5. Predictor-Corrector with Broyden step updates (IN DEVELOPMENT)

- **newtonsCorrection** (*bool, optional*) – If true, a Newtons method correction will be performed after the continuation method is complete.

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,)*) – Concatentation between initial state and Lagrange multiplier for the initial condition constraint,

  z = [x0, lam]

- **T** (*float*) – Current time horizon in the continuation method integration

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

- **dynamics** (*Dynamics object*) – Dynamics system object that can propagate the state and costate to the specified time horizon

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** (*array_like, shape (N+2,)*) – Concatentation between initial state, Lagrange multiplier for the initial condition constraint, and current time horizon in continuation method integration

  z = [x0, lam, T]

- **sig** (*float*) – Current arc length in the pseudo arclength continuation method integration

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

- **dynamics** (*Dynamics object*) – Dynamics system object that can propagate the state and costate to the specified time horizon

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*)

C.8.7 Plotting Module

plotting.**axisEqual**(*ax*, *axesStr='xyz'*)

3D Equivalent of axis equal for matplotlib

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 (x0(T) = x(t0;T)) and final states (xf(T) = x(tf;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. xf_i(T) = x_i(tf;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 booleans de-

termines whether or not the set of possible values for the specified state is shaded gray or not.

- **particleHistBool** (*bool, optional*) – This booleans 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

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

- **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 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 Tf 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

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
- **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

plotting.**plotIntersection**(*R1*, *R2*, *R1color=None*, *R2color=None*, *sc=1.0*, *axesLabels=None*, *axisEqualBool=True*)

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 *particle.xf_T* attribute to create the plot

Parameters

- **R1** (*Reachability object*) – First Reachability object to use in intersection computation
- **R2** (*Reachability object*) – Second Reachability object to use in intersection 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
- **intersectParticles** (*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.

plotting.**plotUnion**(*R1*, *R2*, *R1color=None*, *R2color=None*, *sc=1.0*, *axesLabels=None*, *axisEqualBool=True*)

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

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

C.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 (1,) or (1,1)*) – 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 ap-

proximation. 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.**sympyAbsApprox**(*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 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.**sympySatApprox**(*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 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.**printProgressBar**(*iteration*, *total*, *prefix=''*, *suffix=''*, *decimals=1*, *length=100*, *fill=''*)

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.**null**(*a*, *rtol=2.220446049250313e-16*)

Compute null space of matrix using svd

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 neigh-

bors (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 particles neighbors. Its 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 particles neighbors. Its 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 particles neighbors. Its 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 its 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 vectoras 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 in-

put vector x

Return type sympy expression

REFERENCES

[1] I. M. Mitchell, A. M. Bayen, and C. J. Tomlin, "A time-dependent hamilton-jacobi formulation of reachable sets for continuous dynamic games," *IEEE Transactions on Automatic Control*, vol. 50, no. 7, pp. 947–957, 2005.

[2] M. J. Holzinger, D. J. Scheeres, and R. S. Erwin, "On-orbit operational range computation using gauss's variational equations with J2 perturbations," *Journal of Guidance, Control, and Dynamics*, vol. 37, no. 2, pp. 608–622, 2014.

[3] I. M. Mitchell, M. Chen, and M. Oishi, "Ensuring safety of nonlinear sampled data systems through reachability," *IFAC Proceedings Volumes*, vol. 45, no. 9, pp. 108 –114, 2012, 4th IFAC Conference on Analysis and Design of Hybrid Systems.

[4] P. Varaiya, "Reach set computation using optimal control," in *Proc. KIT Workshop*, 1997, pp. 377–383.

[5] J. Lygeros, "On reachability and minimum cost optimal control," *Automatica*, vol. 40, no. 6, pp. 917–927, Jun. 2004.

[6] A. B. Kurzhanskiy and P. Varaiya, "Dynamic optimization for reachability problems," *Journal of Optimization Theory and Applications*, vol. 108, no. 2, pp. 227–251, 2001.

[7] I. M. Mitchell, "A toolbox of level set methods," UBC Department of Computer Science, Technical Report, 2007.

[8] A. M. Bayen and C. J. Tomlin, "A construction procedure using characteristics for viscosity solutions of the hamilton-jacobi equation," in *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, vol. 2, 2001, 1657–1662 vol.2.

[9] J. Anderson, J. Degroote, G. Degrez, E. Dick, R. Grundmann, and J. Vierendeels, *Computational Fluid Dynamics: An Introduction*, 3rd ed. Springer Berlin Heidelberg, 2009.

[10] I. M. Mitchell and C. J. Tomlin, "Overapproximating reachable sets by hamilton-jacobi projections," *Journal of Scientific Computing*, vol. 19, no. 1, pp. 323–346, 2003.

[11] E. Asarin, T. Dang, and O. Maler, "D/dt: A tool for reachability analysis of continuous and hybrid systems," *IFAC Proceedings Volumes*, vol. 34, no. 6, pp. 741 –746,

2001, 5th IFAC Symposium on Nonlinear Control Systems 2001, St Petersburg, Russia, 4-6 July 2001.

[12] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263, ISBN: 978-3-642-39799-8.

[13] M. Althoff, "An introduction to CORA 2015," in *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, G. Frehse and M. Althoff, Eds., ser. EPiC Series in Computing, vol. 34, EasyChair, 2015, pp. 120–151.

[14] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2E2: A verification tool for stateflow models," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 68–82, ISBN: 978-3-662-46681-0.

[15] I. Hwang, D. M. Stipanović, and C. J. Tomlin, "Polytopic approximations of reachable sets applied to linear dynamic games and a class of nonlinear systems," in *Advances in Control, Communication Networks, and Transportation Systems: In Honor of Pravin Varaiya*. Boston, MA: Birkhäuser Boston, 2005, pp. 3–19, ISBN: 978-0-8176-4409-3.

[16] A. Girard and C. L. Guernic, "Efficient reachability analysis for linear systems using support functions," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 8966–8971, 2008.

[17] J. N. Maidens, S. Kaynama, I. M. Mitchell, M. M. Oishi, and G. A. Dumont, "Lagrangian methods for approximating the viability kernel in high-dimensional systems," *Automatica*, vol. 49, no. 7, pp. 2017 –2029, 2013.

[18] A. B. Kurzhanski and P. Varaiya, "Ellipsoidal techniques for reachability analysis," in *International Workshop on Hybrid Systems: Computation and Control*, Springer, 2000, pp. 202–214.

[19] M. J. Holzinger and D. J. Scheeres, "Reachability set subspace computation for nonlinear systems using sampling methods," in *IEEE Conference on Decision and Control and European Control Conference*, Institute of Electrical and Electronics Engineers (IEEE), 2011.

[20] E. L. Allgower and K. Georg, *Introduction to Numerical Continuation Methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, ISBN: 089871544X.

[21] T. Ohtsuka and H. Fujii, "Stabilized continuation method for solving optimal control problems," *Journal of Guidance, Control, and Dynamics*, vol. 17, no. 5, pp. 950–957, 1994.

[22] N. Baresi, Z. P. Olikara, and D Scheeres, "Fully numerical methods for continuing families of quasi-periodic invariant tori in astrodynamics," vol. 65, Jan. 2018.

[23] C. L. Guernic and A. Girard, "Reachability analysis of linear systems using support functions," *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 250 –262, 2010, IFAC World Congress 2008.

[24] G. Frehse and R. Ray, "Flowpipe-guard intersection for reachability computations with support functions*," *IFAC Proceedings Volumes*, vol. 45, no. 9, pp. 94 –101, 2012, 4th IFAC Conference on Analysis and Design of Hybrid Systems.

[25] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, and R. Grosu, "Xspeed: Accelerating reachability analysis on multi-core processors," in *Hardware and Software: Verification and Testing*, N. Piterman, Ed., Cham: Springer International Publishing, 2015, pp. 3–18, ISBN: 978-3-319-26287-1.

[26] D. Bertsekas, with Angelia Nedic, and A. Ozdaglar, *Convex Analysis and Optimization*. Athena Scientific, 2003, ISBN: 1886529450.

[27] A. Tulsyan and P. I. Barton, "Reachability-based fault detection method for uncertain chemical flow reactors," *IFAC-PapersOnLine*, vol. 49, no. 7, pp. 1 –6, 2016, 11th IFAC Symposium on Dynamics and Control of Process SystemsIncluding Biosystems DYCOPS-CAB 2016.

[28] J. F. Fisac, M. Chen, C. J. Tomlin, and S. S. Sastry, "Reach-avoid problems with time-varying dynamics, targets and constraints," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '15, Seattle, Washington: ACM, 2015, pp. 11–20, ISBN: 978-1-4503-3433-4.

[29] J. Brew, M. J. Holzinger, and S. R. Schuet, "Using reachability to compute unsafe regions in state space through sampling methods," in *International Astronautical Congress*, International Astronautical Federation, 2018.

[30] W. Rudin, *Real and Complex Analysis, 3rd Ed.* New York, NY, USA: McGraw-Hill, Inc., 1987, ISBN: 0070542341.

[31] J. Brew, M. J. Holzinger, and S. R. Schuet, "Reachability subspace exploration using continuation methods," in *AAS/AIAA Space Flight Mechanics Meeting*, 2017.

[32] G. Farin, *Curves and surfaces for computer-aided geometric design: a practical guide*. Elsevier, 2014.

[33] L. Pagani and P. J. Scott, "Curvature based sampling of curves and surfaces," *Computer Aided Geometric Design*, vol. 59, pp. 32 –48, 2018.

[34] M. D. Meyer, P. Georgel, and R. T. Whitaker, "Robust particle systems for curvature dependent sampling of implicit surfaces," in *International Conference on Shape Modeling and Applications 2005 (SMI' 05)*, 2005, pp. 124–133.

[35] A. P. Witkin and P. S. Heckbert, "Using particles to sample and control implicit surfaces," in *ACM SIGGRAPH 2005 Courses*, ACM, 2005, p. 260.

[36] P. Jepp, J. Denzinger, B. Wyvill, and M. C. Sousa, "Using multi-agent systems for sampling and rendering implicit surfaces," in *2008 XXI Brazilian Symposium on Computer Graphics and Image Processing*, 2008, pp. 255–262.

[37] A. Y. Zomaya *et al.*, "Parallel and distributed computing handbook," 1996.

[38] W. Ren and E. Atkins, "Distributed multi-vehicle coordinated control via local information exchange," *International Journal of Robust and Nonlinear Control: IFAC-Affiliated Journal*, vol. 17, no. 10-11, pp. 1002–1033, 2007.

[39] S. Poduri, S. Pattem, B. Krishnamachari, and G. S. Sukhatme, "Using local geometry for tunable topology control in sensor networks," *IEEE Transactions on Mobile Computing*, vol. 8, no. 2, pp. 218–230, 2009.

[40] M. Mesbahi and M. Egerstedt, *Graph theoretic methods in multiagent networks*. Princeton University Press, 2010, vol. 33.

[41] K. Deb and K. Deb, "Multi-objective optimization," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Boston, MA: Springer US, 2014, pp. 403–449, ISBN: 978-1-4614-6940-7.

[42] D. F. Lawden, *Analytical Methods of Optimization*. Mineola, NY: Dover Publications Inc., 2003.

[43] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[44] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and multidisciplinary optimization*, vol. 26, no. 6, pp. 369–395, 2004.

[45] M. G. C. Tapia and C. A. C. Coello, "Applications of multi-objective evolutionary algorithms in economics and finance: A survey," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, IEEE, 2007, pp. 532–539.

[46] Y.-Z. Luo, G.-J. Tang, and Y.-J. Lei, "Optimal multi-objective linearized impulsive rendezvous," *Journal of guidance, control, and dynamics*, vol. 30, no. 2, pp. 383–389, 2007.

[47] P. O. Yapo, H. V. Gupta, and S. Sorooshian, "Multi-objective global optimization for hydrologic models," *Journal of hydrology*, vol. 204, no. 1-4, pp. 83–97, 1998.

[48] I. Das and J. E. Dennis, "Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems," *SIAM journal on optimization*, vol. 8, no. 3, pp. 631–657, 1998.

[49] Y. H. YV, L. S. Lasdon, and D. DA WISMER, "On a bicriterion formation of the problems of integrated system identification and system optimization," *IEEE Transactions on Systems, Man and Cybernetics*, no. 3, pp. 296–297, 1971.

[50] A. Messac, "From dubious construction of objective functions to the application of physical programming," *AIAA journal*, vol. 38, no. 1, pp. 155–163, 2000.

[51] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[52] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[53] C. M. Fonseca and P. J. Fleming, "An overview of evolutionary algorithms in multiobjective optimization," *Evolutionary computation*, vol. 3, no. 1, pp. 1–16, 1995.

[54] J. Rakowska, R. T. Haftka, and L. T. Watson, "Tracing the efficient curve for multi-objective control-structure optimization," *Computing Systems in Engineering*, vol. 2, no. 5-6, pp. 461–471, 1991.

[55] C. Hillermeier, *Nonlinear multiobjective optimization: a generalized homotopy approach*. Springer Science & Business Media, 2001, vol. 135.

[56] O. Schütze, A. Dell'Aere, and M. Dellnitz, "On continuation methods for the numerical treatment of multi-objective optimization problems," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[57] M. J. Daskilewicz, "Methods for parameterizing and exploring pareto frontiers using barycentric coordinates," PhD thesis, Georgia Institute of Technology, 2013.

[58] H. W. Kuhn and A. W. Tucker, "Nonlinear programming," in *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, Calif.: University of California Press, 1951, pp. 481–492.

[59] K. Deb and D. K. Saxena, "On finding pareto-optimal solutions through dimensionality reduction for certain large-dimensional multi-objective optimization problems," Tech. Rep., 2005.

[60] M. Chen, "High dimensional reachability analysis: Addressing the curse of dimensionality in formal verification," PhD thesis, EECS Department, University of California, Berkeley, 2017.

[61] S. Chen, J. Montgomery, and A. Bolufé-Röhler, "Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution," *Applied Intelligence*, vol. 42, Apr. 2015.

[62] A. Martín and O. Schütze, "Pareto tracer: A predictor–corrector method for multi-objective optimization problems," *Engineering Optimization*, vol. 50, no. 3, pp. 516–536, 2018. eprint: https://doi.org/10.1080/0305215X.2017.1327579.

[63] J. Brew, M. Holzinger, and S. Schuet, "Using continuation methods to compute convex reachable volume projections," *submitted to AIAA Journal of Guidance, Control, and Dynamics*, 2019.

[64] ——, "Decentralized techniques for sampling boundary of subspace reachable set," *submitted to AIAA Journal of Guidance, Control, and Dynamics*, 2019.

[65] M. J. Holzinger and D. J. Scheeres, "Reachability results for nonlinear systems with ellipsoidal initial sets," *IEEE transactions on aerospace and electronic systems*, vol. 48, no. 2, pp. 1583–1600, 2012.

[66] A. Gambier and E. Badreddin, "Multi-objective optimal control: An overview," in *2007 IEEE International Conference on Control Applications*, 2007, pp. 170–175.

[67] S. Peitz and M. Dellnitz, "A survey of recent trends in multiobjective optimal control—surrogate models, feedback control and objective reduction," *Mathematical and Computational Applications*, vol. 23, no. 2, p. 30, 2018.

[68] I. M. Mitchell and S. Sastry, "Continuous path planning with multiple constraints," in *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, vol. 5, 2003, 5502–5507 Vol.5.

[69] A. Kumar and A. Vladimirsky, "An efficient method for multiobjective optimal control and optimal control subject to integral constraints," *Journal of Computational Mathematics*, pp. 517–551, 2010.

[70] M. J. Holzinger, D. J. Scheeres, and J. Hauser, "Reachability using arbitrary performance indices," *IEEE Transactions on Automatic Control*, vol. 60, no. 4, pp. 1099–1103, 2015.

[71] M. J. Holzinger, D. J. Scheeres, and J. Hauser, "Optimal reachability sets using generalized independent parameters," in *Proceedings of the 2011 American Control Conference*, 2011, pp. 905–912.

[72] A. E. Bryson and Y. C. Ho, *Applied Optimal Control*. New York: Blaisdell, 1969.

[73] I. M. Mitchell, "Comparing forward and backward reachability as tools for safety analysis," in *Hybrid Systems: Computation and Control*, A. Bemporad, A. Bicchi, and G. Buttazzo, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 428–443, ISBN: 978-3-540-71493-4.

[74] K. Sundaresan, "Smooth banach spaces," *Bull. Amer. Math. Soc.*, vol. 72, no. 3, pp. 520–521, May 1966.

[75] H. Sussmann, "A general theorem on local controllability," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 158–194, 1987.

[76] J. C. Butcher, *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[77] R. M. Mattheij, "The conditioning of linear boundary value problems," *SIAM Journal on Numerical Analysis*, vol. 19, no. 5, pp. 963–978, 1982.

[78] L. N. Trefethen and D. B. III, *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[79] D. Ruiz, "A scaling algorithm to equilibrate both rows and columns norms in matrices," Tech. Rep., 2001.

[80] H. Schaub and J. L. Junkins, *Analytical Mechanics of Space Systems*. Reston, VA: AIAA Education Series, 2003.

[81] R. Beard, "Quadrotor dynamics and control," Apr. 2019.

[82] F. Immler, M. Althoff, X. Chen, C. Fan, G. Frehse, N. Kochdumper, Y. Li, S. Mitra, M. S. Tomar, and M. Zamani, "Arch-comp18 category report: Continuous and hybrid systems with nonlinear dynamics," in *ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems*, G. Frehse, Ed., ser. EPiC Series in Computing, vol. 54, EasyChair, 2018, pp. 53–70.

[83] J. Gravesen, "Surfaces parametrized by the normals," *Computing*, vol. 79, no. 2, pp. 175–183, 2007.

[84] M. L. Sampoli and B. Jüttler, "Support function representation for curvature dependent surface sampling," in *Applied and Industrial Mathematics in Italy III*. World Scientific, 2009, pp. 520–531. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789814280303_0046.

[85] M. E. Muller, "A note on a method for generating points uniformly on n-dimensional spheres," *Commun. ACM*, vol. 2, no. 4, pp. 19–20, Apr. 1959.

[86] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997, ISBN: 0-201-89683-4.

[87] X. Ge, F. Yang, and Q.-L. Han, "Distributed networked control systems: A brief overview," *Information Sciences*, vol. 380, pp. 117–131, 2017.

[88] F. Bullo, J. Cortes, and S. Martinez, *Distributed control of robotic networks: a mathematical approach to motion coordination algorithms*. Princeton University Press, 2009, vol. 27.

[89] N. E. Leonard and E. Fiorelli, "Virtual leaders, artificial potentials and coordinated control of groups," in *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, IEEE, vol. 3, 2001, pp. 2968–2973.

[90] P. Ogren, E. Fiorelli, and N. E. Leonard, "Cooperative control of mobile sensor networks: Adaptive gradient climbing in a distributed environment," *IEEE Transactions on Automatic control*, vol. 49, no. 8, pp. 1292–1302, 2004.

[91] Y. Hong, G. Chen, and L. Bushnell, "Distributed observers design for leader-following control of multi-agent networks," *Automatica*, vol. 44, no. 3, pp. 846–850, 2008.

[92] M. A. Batalin and G. S. Sukhatme, "Spreading out: A local approach to multi-robot coverage," in *Distributed Autonomous Robotic Systems 5*, Springer, 2002, pp. 373–382.

[93] C. H. Caicedo-Nunez and M. Zefran, "A coverage algorithm for a class of non-convex regions," in *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, IEEE, 2008, pp. 4244–4249.

[94] L. Meirovitch, *Methods of Analytical Dynamics*, ser. Dover Civil and Mechanical Engineering. Dover Publications, 2012, ISBN: 9780486137599.

[95] R. K I, G Deepa, and D. K. Namboori, *Computational Chemistry and Molecular Modeling- Principles and applications K.I.Ramachandran, G.Deepa and Krishnan Namboori P.K.-Springer-Verlag GmbH, Germany: Springer International (2008).* Jan. 2008.

[96] D. Hardin and E. Saff, "Discretizing manifolds via minimum energy points," *Notices of the AMS*, vol. 51, no. 10, pp. 1186–1194, 2004.

[97] W. M. Haddad and V. Chellaboina, *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2008.

[98] *The Finite Element Method for Solid and Structural Mechanics*. Elsevier, 2014.

[99] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE Transactions on Automatic Control*, vol. 31, no. 9, pp. 803–812, 1986.

[100] D. Bertsekas, *Nonlinear Programming*. Athena Scientific, 1999.

[101] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," *SIGMOD Rec.*, vol. 24, no. 2, pp. 71–79, May 1995.

[102] P.-A. Absil, R. E. Mahony, and B. Andrews, "Convergence of the iterates of descent methods for analytic cost functions," *SIAM Journal on Optimization*, vol. 16, pp. 531–547, 2005.

[103] G. A. Young and R. L. Smith, *Essentials of Statistical Inference*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2005.

[104] E. B. Dam, M. Koch, and M. Lillholm, *Quaternions, interpolation and animation*. Citeseer, 1998, vol. 2.

[105] D. Liberzon, *Calculus of Variations and Optimal Control Theory: A Concise Introduction*. Princeton University Press, 2012.

[106] P. L. Yu, "Cone convexity, cone extreme points, and nondominated solutions in decision problems with multiobjectives," *Journal of Optimization Theory and Applications*, vol. 14, no. 3, pp. 319–377, 1974.

[107] M. Sakawa and H. Yano, "Generalized hyperplane methods for characterizing lambda-extreme points and trade-off rates for multiobjective optimization problems," *European Journal of Operational Research*, vol. 57, no. 3, pp. 368 –380, 1992.

[108] J. M. Longuski, J. J. Guzmán, and J. E. Prussing, *Optimal Control with Aerospace Applications*. Springer Nature, 2014.

[109] R. F. Stengel, *Optimal Control and Estimation*, ser. Dover Books on Mathematics. Dover Publications, 1994.

[110] E. H. Spanier, *Algebraic topology*, 1. Springer Science & Business Media, 1989, vol. 55.

[111] L. T. Watson, "Numerical linear algebra aspects of globally convergent homotopy methods," *SIAM Review*, vol. 28, no. 4, pp. 529–545, 1986.

[112] W. C. Rheinboldt, "Numerical continuation methods: A perspective," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1, pp. 229 –244, 2000, Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[113] J. S. Arora, "Chapter 18 - multi-objective optimum design concepts and methods," in *Introduction to Optimum Design (Fourth Edition)*, J. S. Arora, Ed., Fourth Edition, Boston: Academic Press, 2017, pp. 771 –794, ISBN: 978-0-12-800806-5.

[114] S. Bansal, M. Chen, S. L. Herbert, and C. J. Tomlin, "Hamilton-jacobi reachability: A brief overview and recent advances," *CoRR*, vol. abs/1709.07523, 2017. arXiv: 1709.07523.

[115] B. HomChaudhuri, M. Oishi, M. Shubert, M. Baldwin, and R. S. Erwin, "Computing reach-avoid sets for space vehicle docking under continuous thrust," in *2016 IEEE 55th Conference on Decision and Control (CDC)*, 2016, pp. 3312–3318.

[116] J. Darbon and S. Osher, "Algorithms for overcoming the curse of dimensionality for certain hamilton–jacobi equations arising in control theory and elsewhere," *Research in the Mathematical Sciences*, vol. 3, no. 1, 2016.

[117] M. J. Holzinger and D. J. Scheeres, "Reachability results for nonlinear systems with ellipsoidal initial sets," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 48, no. 2, pp. 1583–1600, 2012.

[118] A. Messac, G. J. Sundararaj, R. V. Tappeta, and J. E. Renaud, "Ability of objective functions to generate points on nonconvex pareto frontiers," *AIAA Journal*, vol. 38, no. 6, pp. 1084–1091, 2000. eprint: https://doi.org/10.2514/2.1071.

[119] C. W. Warren, "Global path planning using artificial potential fields," in *Proceedings, 1989 International Conference on Robotics and Automation*, Ieee, 1989, pp. 316–321.

[120] S. Prajna and A. Jadbabaie, "Safety verification of hybrid systems using barrier certificates," in *International Workshop on Hybrid Systems: Computation and Control*, Springer, 2004, pp. 477–492.

[121] A. A. Kurzhanskiy and P. Varaiya, "Ellipsoidal techniques for reachability analysis of discrete-time linear systems," *IEEE Transactions on Automatic Control*, vol. 52, no. 1, pp. 26–38, 2007.

[122] P.-J. Meyer, A. Devonport, and M. Arcak, "Tira: Toolbox for interval reachability analysis," in *Proceedings of the 22Nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19, Montreal, Quebec, Canada: ACM, 2019, pp. 224–229, ISBN: 978-1-4503-6282-5.

[123] B. Krauskopf, H. M. Osinga, and J. Galán-Vioque, *Numerical continuation methods for dynamical systems*. Springer, 2007.

[124] D. Folta and F. Vaughn, "A survey of earth-moon libration orbits: Stationkeeping strategies and intra-orbit transfers," in *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, 2004, p. 4741.

[125] W. S. Koon, M. W. Lo, J. E. Marsden, and S. D. Ross, "Low energy transfer to the moon," *Celestial Mechanics and Dynamical Astronomy*, vol. 81, no. 1-2, pp. 63–73, 2001.

[126] R. W. Farquhar, D. W. Dunham, Y. Guo, and J. V. McAdams, "Utilization of libration points for human exploration in the sun–earth–moon system and beyond," *Acta Astronautica*, vol. 55, no. 3-9, pp. 687–700, 2004.

[127] N. Bosanac, A. D. Cox, K. C. Howell, and D. C. Folta, "Trajectory design for a cislunar cubesat leveraging dynamical systems techniques: The lunar icecube mission," *Acta Astronautica*, vol. 144, pp. 283 –296, 2018.

[128] M. Althoff, S. Bak, X. Chen, C. Fan, M. Forets, G. Frehse, N. Kochdumper, Y. Li, S. Mitra, R. Ray, C. Schilling, and S. Schupp, "Arch-comp18 category report: Continuous and hybrid systems with linear continuous dynamics," in *ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems*, G. Frehse, Ed., ser. EPiC Series in Computing, vol. 54, EasyChair, 2018, pp. 23–52.

[129] D. Eppstein, "Zonohedra and zonotopes," Online, 2019.