

**Coho: A Verification Tool for Circuit Verification by
Reachability Analysis**

by

Chao Yan

B.S., Peking University, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

August 2006

© Chao Yan, 2006

Abstract

COHO is a verification tool for systems modeled by nonlinear ordinary differential equations (ODEs). Verification is performed using reachability analysis. The reachable space is represented by projectagons which are the polyhedron described by their projection onto two dimensional subspace. COHO approximates nonlinear ODEs with linear differential inclusions to compute the reachable state.

We re-engineer the COHO system to provide a robust implementation with a clear interface and well-organized structure. We demonstrate the soundness and robustness of our approach by applying it to several examples, including a three dimensional, non-linear systems for which previous version of COHO failed due to numerical stability problems.

The correctness of COHO strongly depends on the accuracy, efficiency and robustness of the linear program solver that is used throughout the analysis. Our COHO linear program solver is implemented by integrating the Simplex algorithm with interval arithmetic based on the framework set up by Laza [Laz01]. For highly ill-conditioned problems, an approximation that is very close to the optimal result is computed by our algorithm. This results in a very small error bound that allows COHO to successfully verify interesting examples.

This thesis also presents an algorithm to solve the problem of projecting the feasible region of a linear program onto two dimensional subspaces. This algorithm uses the COHO linear program solver for efficiency and accuracy. We derive an analytical upper bound for the error and present experimental results to show that the errors are negligible in practice.

Contents

Abstract	ii
Contents	iii
List of Tables	vi
List of Figures	vii
Acknowledgments	ix
Dedication	x
1 Introduction	1
1.1 Motivation	1
1.2 Contribution	3
1.3 Thesis Organization	4
2 Background and Related Work	7
2.1 Formal Verification	7
2.2 Verification as Reachability	10
2.3 Projectagons	12
2.4 COHO	14
3 Architecture of the Coho System	23

3.1	Java Component	24
3.2	Number Package	25
3.3	Matrix Package	29
4	Coho Linear Program Solver	32
4.1	Linear Programming and the Simplex Algorithm	33
4.1.1	COHO Linear Programming	33
4.1.2	Primal-Dual Method	34
4.1.3	Simplex Algorithm	35
4.2	Combination of Simplex and Interval Computation	36
4.2.1	Ambiguous Pivoting and Cycling	38
4.2.2	Possible Optimal Result	40
4.2.3	Initial Feasible Basis	40
4.3	Efficient Linear System Solver	42
4.4	Ill-Conditioned Basis and Exception Handling	44
4.4.1	Conditioning	44
4.4.2	Use of Non-optimal Bases	47
4.5	Implementation	50
5	Projection	53
5.1	Algorithm	53
5.2	Implementation	57
5.3	Converting from End-of-Step to Beginning-of-Step Projection	61
5.4	Error Analysis	64
6	Experiments	69
6.1	Application of COHO Verification System	69
6.2	Experimental Result for LP solver and LP project	74

7 Conclusion	80
7.1 What has been Accomplished	80
7.2 Suggestions for Further Research	81
Bibliography	86

List of Tables

2.1	Comparison of Hybrid System Verification Tools	9
3.1	Functions of CohoDoubleInterval	29

List of Figures

2.1	A Three Dimensional “Projectagon”	13
2.2	A Time Step of COHO	16
2.3	The Creation of a Bloated Linear Program from Projectagon	17
3.1	The Architecture of COHO System	23
3.2	Package Hierarchy of COHO System	25
3.3	Class Hierarchy of Number Package	26
3.4	Class Hierarchy of Matrix Package	31
4.1	Pivot Selection of COHO Linear Program Solver	39
4.2	COHO Linear System Solver	44
4.3	Types of Optimal 2D Vertices	49
4.4	Highly Obtuse Optimal 3D Vertices	50
4.5	Class Hierarchy of COHO Linear Program Solver	51
5.1	Projection Algorithm Using Linear Programming	54
5.2	Find the Normal of Current Edge Using Optimal Basis of Previous Linear Program	55
5.3	Overestimation of Projection Polygon When an Edge is Skipped	57
5.4	The Representation of Optimization Direction	57
5.5	Find the Initial Optimization Direction for Projection Algorithm	58
5.6	Backtracking to Reduce Error When Jump to Next Quadrant	60

5.7	Special Case of Jumping Two Quadrants Continuously	61
5.8	Conversion between End-of-Step Projection and Beginning-of-Step Projection	63
5.9	Projection Error Introduced by COHO Linear Program Solver	65
5.10	Projection Error Introduced by Ignoring Small Angle	66
5.11	Projection Error Introduced by Skipping Short Edge	66
5.12	Projection Error Introduced by Jumping to Next Quadrant	67
6.1	The Result of a Two Dimensional Linear Model Example: Sink	70
6.2	The Result of 3VdP Example: x - y plane	73
6.3	The Result of 3VdP Example: y - z plane	73
6.4	Error Distribution of Optimal Result of COHO Linear Program Solver	75
6.5	Distribution Number of Branches of a Pivot	76
6.6	Distribution of Condition Number for Ill-Conditioned Problems	77
6.7	Number of Linear Programs and Exceptions per Step	78

Acknowledgments

Without encouragement from Mark Greenstreet, it would be impossible to write this thesis. It is difficult to express my gratitude for his extensive support, discussion and endless intelligent ideas.

I am grateful to Ian Mitchell, Chen Greif, and Michael Friedlander for their contribution of time and ideas to my work.

I would also like to thank Marius Laza, Ian Mitchell and many others I do not know their names for their previous hard work on COHO. Without their work, it is impossible to complete COHO by myself.

Special thanks to my parents and yuliang for too many reasons to say.

I would like to thank my friends, Suwen Yang, Xiushan Feng, Suling Yang, Xun Sun, Wei Li, Xiang Xuan, Fei Ma, Shu'an Wang, Jie Zhao, Julia Chen, Chen Yang and many others, for enjoying the happy life together.

CHAO YAN

*The University of British Columbia
August 2006*

To my love, Yuliang.

Chapter 1

Introduction

1.1 Motivation

With the increasing of the complexity of circuit design, it is more and more important to verify that a circuit satisfies its high-level specification automatically and formally. Prototyping and simulation are two means of validation. However, neither gives full assurance that all possible situations have been checked and the system always works as expected. Furthermore, compared with increasing of design complexity, the percentage of problem cases covered by manually generated test suites or pseudo-random functional testing is declining. Therefore, formal verification, a mathematical approach that exhaustively proves functions properties, has attracted more and more attention.

For digital circuits or hybrid systems, which contain both discrete components and continuous components, the verification problem can be formulated as reachability analysis. The system is modeled by some mathematical model, such as ordinary differential equations (ODEs). Thus, verification amounts to computing the set of reachable states and checking the properties on such states.

COHO, a verification tool for circuit design, performs reachability analysis. It models circuits by ODEs and represents reachable regions by their projection onto two-dimensional subspaces. Generally, the nonlinear ODE model does not

have closed form solutions, and approximation techniques must be used. COHO computes an over approximation of the reachable space such that it might fail to verify a correct system, but it never mistakenly verifies an incorrect design.

To compute the evolution of reachable states, COHO solves a large number of linear programs. The accuracy and efficiency of the linear program solver plays a great role for the correctness and performance of COHO. Linear programming is a well-know problem with classical solutions such as the Simplex algorithm. Exploiting the special structure of linear programs that arise from COHO, Laza [Laz01] presented an efficient $O(n)$ linear system solver for Simplex algorithm such that the optimal result is accurate and the linear program solver is efficient.

Floating point computation does not guarantee that the optimal value for the linear program is always over approximated as required. An under approximation of the error might make it possible that COHO verifies an incorrect design. Furthermore, the soundness of COHO relies heavily on the accuracy of the optimal result of linear programs. Therefore the linear program solver is required to produce solutions as accurately as possible, especially for the badly conditioned problems that often arise in the COHO's analysis. Otherwise, the large over-approximation error prevents COHO from verifying correct systems.

These problems are solved by implementing the Simplex algorithm with interval arithmetic which provides both the result and its error bound. Interval computation guarantees over approximation of the final result. Ill-conditioned problems are easily detected by the interval representation. Building on the interval arithmetic methods, we present a new algorithm for finding a good approximation of the optimal result. Using the COHO linear program solver, we also developed an algorithm to compute the projection of a polyhedron onto a two-dimensional subspace. We demonstrate these capabilities of these new algorithms by applying them to the analysis of a simple, non-linear system which could not be analysed by prior versions of COHO.

1.2 Contribution

This research focused on improving and implementing a robust and efficient linear program solver based on previous work [Laz01] and developing a new projection algorithm for COHO. This improvement and development are complemented by a re-engineering of the COHO source code to produce a robust implementation of the COHO system.

The main contributions of this thesis include:

1. The re-engineering of the COHO software.
 - We defined general purpose interfaces for numeric types and matrices. These provided an easily extended framework for integrating interval arithmetic into COHO and implementing the linear program solver and projection algorithms.
 - Interval arithmetic is integrated into both linear program solver and projection algorithms.
 - Re-engineering of the linear program solver to improve its modularity. The various phases of the Simplex algorithm are clearly separated, and we make systematic use of exceptions for handling very badly conditioned bases.
2. Improvement and reimplementation of the linear program solver using interval computation
 - The solver provides both upper bounds and lower bounds which guarantee overapproximation as required by COHO, and the error bound is quite small.
 - We developed a much more efficient algorithm to estimate the condition number of a basis than the previous method.

- We implemented the method proposed in [Laz01] for discarding (nearly) redundant constraints to handle very badly conditioned bases.
 - We took special care to eliminate all tests for floating-point equality from these algorithms. This corrected several errors that remained in the previous implementation.
3. An implementation of a specialized algorithm for projecting the feasible polyhedron onto a two-dimensional subspace.
 - Using our COHO solver, the projection method is efficient.
 - The algorithm over approximates the projection polygon with small error.
 4. Apply the revised COHO system to two examples:
 - A simple, sink with linear dynamics from [DM98]. We show that our implementation of COHO provides much tighter bounds than those previously reported for other tools.
 - A van der Pol oscillator with three state variables. This uses the projection capability of COHO. Furthermore, the previous version of COHO was unable to verify this example due to numerical stability issues in the linear program solver that we have now overcome.

1.3 Thesis Organization

This thesis consists of seven chapters as described below:

- Chapter 2 introduces the COHO system in which the linear program solver and projection algorithm are applied. First, reachability analysis and its application to verification, especially circuit verification are introduced. Then the concept of representing reachable space by projections is described briefly. Finally, the algorithm and structure of the COHO verification system is presented.

- Chapter 3 presents the architecture of the COHO system. The partition a Matlab subsystem and a Java subsystem is explained first, followed by the description of packages that we wrote for interval arithmetic and matrices.
- Chapter 4 presents the implementation of the COHO linear program solver. We first give a brief introduction to linear programming and the special structure of the linear programs that arise in COHO. Then the problems of integrating Simplex with interval arithmetic are presented, followed with our solutions. To improve the speed of COHO solver, Laza's linear system solver is employed and improved. The algorithm to find a good approximation of the optimal cost when the optimal basis is badly conditioned is also presented with geometric explanations. The chapter ends with a description of some implementation issues.
- Chapter 5 presents our algorithm for the projection problem. The basic idea is that we can use the feasibility of the current basis to find the next edge. We then describe implementation problems and solutions. Moving the reachable region forward in time according to its differential inclusion model produces a linear program with a different structure than we have assumed so far. Fortunately, this end-of-time-step linear program can be expressed as the product of a linear program with the special COHO structure and an easily invertible matrix. We then extend our projection algorithm to work with linear programs of this form.

Finally, the error of our projection algorithm is analyzed, and an upper bound on the error is obtained.

- Chapter 6 presents experimental result of COHO system. We first describe two examples and present the results produced by the COHO verification tool. Then the experimental results for COHO linear program solver and projection algorithm are examined which demonstrate the robustness of our algorithms.

- A chapter of conclusions and research topics in the future completes this thesis.

Chapter 2

Background and Related Work

This chapter describes related research in hybrid system verification and prior work on the COHO system. Section 2.1 summarizes existing approaches for the formal verification of hybrid systems. Sections 2.2 through 2.4 describe the COHO system. Section 2.2 describes how COHO formulates the verification problem for circuits and hybrid systems as a reachability problem for systems modeled by differential inclusions. A critical issue in such analysis is finding a tractable representation for high dimensional objects. COHO uses “projectagons”, where the reachable space is represented by its projection onto two dimensional subspaces. Projectagons are introduced in section 2.3. Then, Section 2.4 describes how algorithm that COHO uses to compute the reachable space for a circuit or hybrid system.

2.1 Formal Verification

Formal verification endeavors to prove or disprove that a system satisfies a certain formal specification or property using formal methods from mathematics. The design might be a hardware system, a software system, a network protocol, an airplane, etc. To verify such systems, formal, abstract mathematical models must be constructed for both the system and the specification. Commonly used mathematical models include finite state machine (FSM), labelled transition systems, Petri

nets, etc.

Recently, formal verification for discrete systems have gained industrial success, especially in the areas of digital circuits and communication protocols. Extending formal verification techniques to systems with continuous dynamics has been a particularly challenging research field.

Hybrid Systems are dynamical systems with both discrete and continuous behaviours. For example, an embedded system, a flight controller and a circuit with both digital and analog signals are all hybrid systems. Verifying hybrid systems formally is increasingly important because of the growing prevalence of computers as controllers in safety critical systems and the importance of analog circuit effects in deep-submicron integrated circuits.

A commonly used formal model for hybrid systems is the hybrid automaton [Hen96]. Verification problems for most classes of systems with nonlinear continuous dynamics are undecidable. A common approach to verification of such systems thus involves some type of approximate reachability calculation.

Many verification tools have been developed for hybrid systems. Table 2.1 summarizes four of the most widely used tools that are closely related to COHO.

HyTech [HHWT97] [HPWT01] is a symbolic model checker for *linear hybrid automata* [Hen96]. The hybrid system is modeled as a collection of automata with discrete and continuous components, and the properties to verify are expressed using temporal logic formulas. HyTech can check safety properties; if the verification fails, HyTech generates a diagnostic error trace. Another key feature of HyTech is its ability for parametric analysis to determine necessary and sufficient constraints on the parameters under which the system is safe.

d/dt [ADM02] is a tool for reachability analysis of continuous or hybrid systems using linear differential inclusions of the form of $dx/dt = Ax + Bu$, where u is an external input taking values in a bounded convex polyhedron. *d/dt* represents the reachable sets as non-convex orthogonal polyhedra [BMP99],

Table 2.1: Comparison of Hybrid System Verification Tools

Tool	HyTech	d/dt	CheckMate	PHAVer
Model	Linear hybrid automata	Hybrid automata with linear differential inclusion	Polyhedral invariant hybrid automata	Linear hybrid automata
CTL specification	yes	no	ACTL	no
Representation of Reachable Set	Hyper rectangular	Orthogonal polyhedron	Polyhedron defined by $Cx \leq d$	Polyhedron defined by a set of constraints
Approximation Technique	1) clock translation 2) rate translation 3) linear phase-protrait approximation	face lifting [DM98]	1) quotient transition systems (QTS) 2) flowpipe approximation	On-the-Fly over-approximation of piecewise affine dynamics
Counter Example	yes	no	yes	no

i.e. finite unions of full-dimensional hyper-rectangles, and approximates the reachable state using numerical integration and polyhedral approximation. d/dt can be used to compute reachable sets, verify safety properties of systems, and synthesize safety properties of switching controllers based on the *maximal invariant set*.

CheckMate [SR⁺00] is a Matlab based tool for modeling, simulating and verifying properties of a class of hybrid systems: *threshold-event-driven hybrid systems*. The system is represented as a *polyhedral invariant hybrid automaton (PIHA)*, with three types of dynamics: clock dynamics ($dx/dt = c$), linear dynamics ($dx/dt = Ax + b$) and nonlinear dynamics ($dx/dt = f(x)$). Therefore, verification can be performed directly on a model of the hybrid system without constructing a linear approximation [SK00], as HyTech and d/dt do. The system specification is expressed as an ACTL formula, a restriction of Computational Tree Logic (CTL) [Eme97]. CheckMate can perform two kinds of verifications: a quick verification and a complete verification. With the quick

verification, only the vertices of the polyhedron defined by the initial conditions are verified. The advantages of the quick verification is that it is fast and can provide a rough idea about the final result of the complete verification. If the quick verification finds an error, then the design violates the specification and should be corrected. However, the quick verification may fail to find some errors. Thus if the quick verification succeeds, then a complete verification can be performed to make sure that the design is correct.

PHAVer [Fre05] is a verification tool for linear hybrid automata that supports compositional and assume-guarantee reasoning. It uses the Parma Polyhedra Library that supports arbitrary large number, thus it provides exact, robust arithmetic. The system model is based on affine dynamics that are handled by on-the-fly overapproximation.

Table 3.1 compares some important features of these tools. There is another hybrid system verification tools that deserves some attention: *VeriShift*. In contrast to all tools described above, it approximates the reachable region using ellipsoids [BT00]. The advantages of ellipsoidal approximation are a reduced memory requirement and a polynomial complexity of ellipsoidal operations.

2.2 Verification as Reachability

To verify a system, the formal model is checked for correctness with respect to the formally expressed specification. The entire reachable state space of the system should be explored. Representing and computing these reachable states is the essential step for many verification tasks.

Given a n -dimensional region $A \in R^n$, the reachability problem is to compute the (forward) reachable state space that contains all trajectories that start in A according the model of the system, either during some time interval $[0, t_{end}]$ or for all time. Generally, the model can be represented by an ordinary differential

inclusion:

$$\dot{x} \in F(x, t) \tag{2.1}$$

where $x \in R^n$. The inclusion in the model allows uncertainty.

Many verification problems can be solved as reachability analysis problems. For example, the safety property verification problem is to explore the entire reachable states space of system and validate the property on all the reachable states.

Reachability analysis can be used to verify whether a circuit correctly implements its specification. Generally, a circuit can be modeled by ordinary differential inclusions (ODIs), and the initial state and constraints determine the initial region in the state space. Then, reachable sets are computed by advancing the initial region according to the ODI model. After exploring all the reachable states, the circuit verification problem becomes a problem of checking whether the specification is valid for all reachable states.

Unfortunately, closed form solutions do not exist for general ODI models, except for some special cases such as linear models. Thus, an approximation based approach must be applied for reachability analysis. For verification problems, the reachable state space should be over approximated. It is sound that incorrect systems are never falsely verified, although the over approximation might make it fail to verify a correct system. In COHO the reachable space at each time step is partitioned into small pieces, and ODI model for each piece is approximated by a linear differential inclusion:

$$F(x, t) \subseteq A \cdot x + b + \text{cube}(u) \tag{2.2}$$

where $\text{cube}(u)$ is a hypercube, i.e., the Cartesian product of n intervals, which represents an error bound for the approximation.

2.3 Projectagons

The representation of the reachable space strongly affects the efficiency and accuracy of a reachability analysis algorithm.

For example, approximating the reachable space by the minimum hyperrectangle that contains it is quite simple. The number of vertices/faces of hyperrectangles increases linearly with the dimension, and operations on hyperrectangles are also quite efficient. However, hyperrectangles produce an unacceptably large over approximation of the reachable space for most applications. At the other extreme, general nonconvex high-dimensional polyhedra can represent arbitrary, high-dimensional objects with relatively small error. However, manipulations of general n dimensional polyhedra typically have time and space complexities with exponents of n or $n/2$.

In COHO, reachable sets are represented as *projectagons*. For the purposes of this thesis, a projectagon is the high dimensional (and potentially nonconvex) bounded polygon formed by the intersection of a collection of prisms. Each prism is unbounded in all but two dimensions, and in those two dimensions the cross-section of the prism is a bounded polygon. By choosing this representation we need only track the two-dimensional cross-sections of the prisms rather than any full dimensional object. We take advantage of this simplifying property in both the computational geometry and numerical algorithms. For example, Figure 2.1 shows how a three-dimensional object (the “anvil”) can be represented by its projection onto the xy , yz , and xz planes.

The projection polygons are not required to be convex; thus, non-convex, high-dimensional objects can be represented by projectagon. The high-dimensional object represented by a projectagon is the largest set of points that satisfies the constraints of each projection. It can be obtained from its projections by back-projecting each projection polygon into a prism in R^n and computing the intersection of these prisms. This can lead to an overapproximation of the polyhedron. Thus,

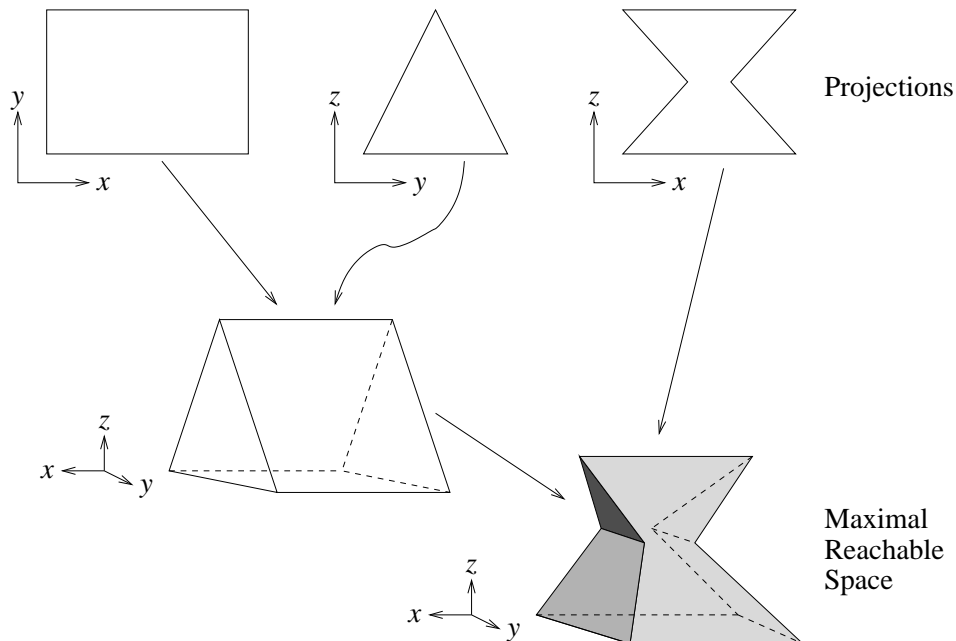


Figure 2.1: A Three Dimensional “Projectagon” [Laz01, Figure 2.1]

not all polyhedra can be represented exactly using projectagons. For example, the projectagon representation of an object with one or more indentations on its faces is an projectagon that has these indentations filled in.

Projectagons offer several advantages. Because projectagons are based on two-dimensional polygons, they can be efficiently manipulated using well-known algorithms from computational geometry [PS85]. Ignoring degeneracies, faces of the object represented by a projectagon correspond to edges of its projection polygons. Assuming that the model is linear and has finite derivatives, the extremal trajectories are those emanating from faces. The projectagon representation allows many operations on faces including calculations of their trajectories to be carried out as simple operations on polygon edges.

Many other representations have been used in reachability analysis tools. Bournez and Maler [BMP99] investigates *orthogonal polyhedra*, which partition polyhedra into finite unions of full dimensional hyper-rectangles. This representation is

canonical for all polyhedra, convex or non-convex, in any dimension. Geometric operations on orthogonal computations are efficient. They focus on a special case that they call *griddy polyhedra* where the vertices of the polyhedron are constrained to lie on evenly spaced gridpoints in R^d . [SK03] presents a method that approximates reachable sets using *oriented rectangular hulls* (ORHs). An ORH is a hyperrectangle whose orientation is determined by the reachable set rather than being forced to be parallel to the coordinate axes of the model. A preferred orientation is derived from the singular value decomposition (SVD) of sample covariance matrices for sets of reachable states. The orientations keep the over-approximation of the reachable sets small in most cases with a complexity of low polynomial order with respect to the dimension of the continuous reachable state space. As mentioned in section 2.1, ellipsoidal techniques are also attractive because they use only $O(n^2)$ space for n dimensional object, and the complexity of ellipsoidal operations is also polynomial rather than exponential for general polyhedral operations. However, the disadvantage is that the intersection or union of ellipsoids are no longer ellipsoids, and thus approximations must be performed.

2.4 Coho

COHO [GM98, GM99] is a reachability analysis tool for systems modeled by non-linear ordinary differential equations. COHO uses projectagons to represent reachable regions, and over approximates the ordinary differential equations model by linear differential inclusion, as described in section 2.2 and 2.3.

As described in the previous section, the projectagon representation allows many operations on faces including calculations of their trajectories to be carried out as simple operations on polygon edges. Therefore, COHO advances the projectagon of a reachable set by considering each face of the projectagon in turn. COHO moves each edge/face forward in time with an over approximation obtained by a simple Euler integrator [RLB01]. Although an Euler integrator is less accurate

than higher-order methods, its simplicity allows us to guarantee that our results are overapproximations as desired.

As illustrated in Figure 2.2 from [Laz01], COHO advances the reachable region by performing the following operations at each time step (see [GM99] for details):

1. Bloat the projectagon

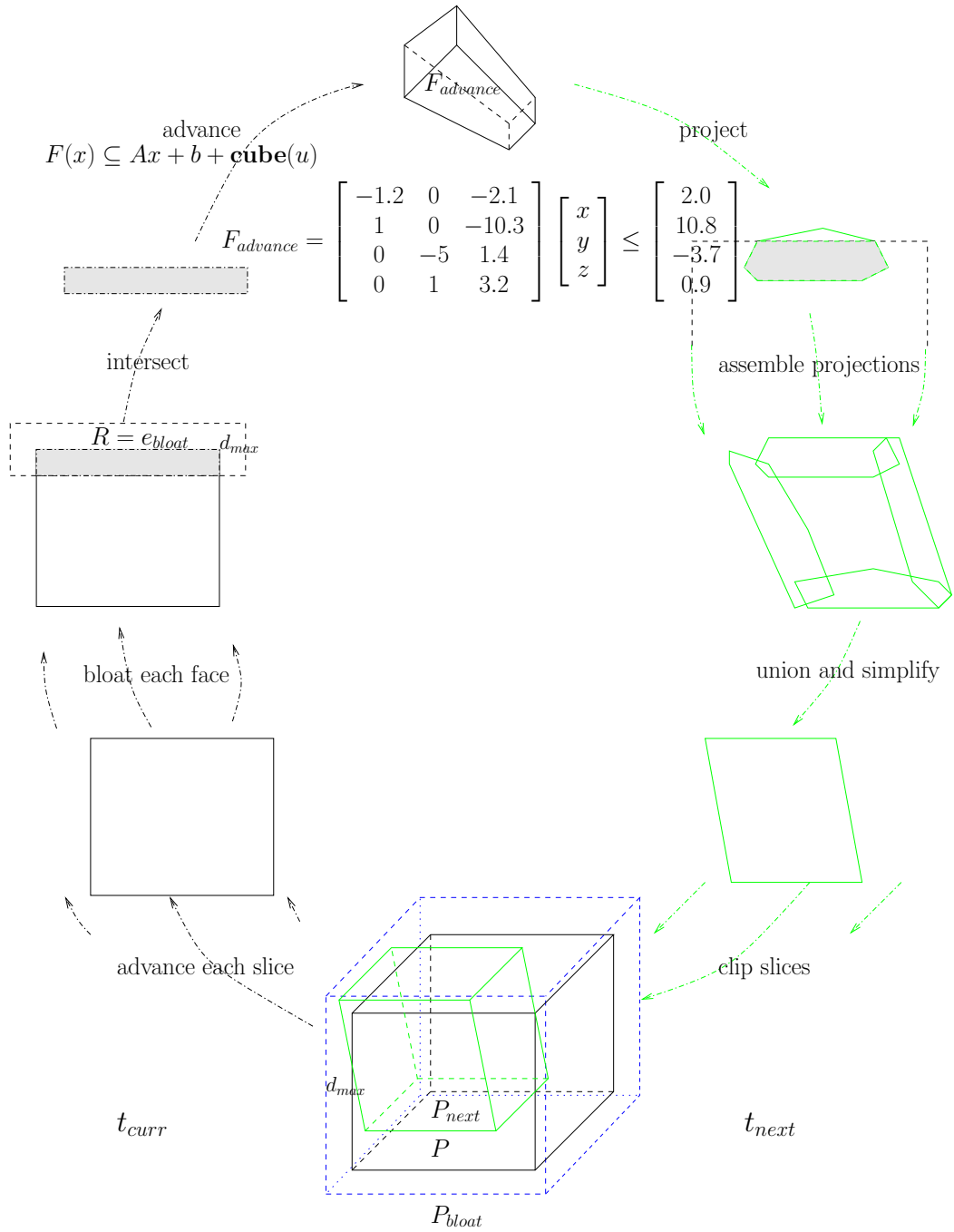
Let P be a projectagon. To compute all points reachable from P in the next times step, we first compute H , an possibly over approximation of the convex hull of P , and then B , a bloated version of H . H is simply the projectagon corresponding whose projection polygons are the convex hulls of the projection polygons of P ; in other words, by approximate the convex hull of P by computing the hulls of the each of its projection polygons independently. We can describe H as a linear program: $Ax \leq c$. We normalize A such that each row of A has a l_2 norm of 1. Now, the linear program for B is $Ax \leq c + d_{bloat}$, where d_{bloat} is the bloating distance. Figure 2.3 illustrates this process. In the following, we refer to a projection polygon of a projectagon as a *slice*.

2. Linearize the model for each edge

For each edge, e , COHO creates a rectangle, R_e whose edges are either parallel or perpendicular to e such that R_e contains e with a separation of d_{bloat} . Clearly, R_e can be represented by a linear program; thus, we will write R_e to denote this linear program in the following. R_e is used to compute the step size such that trajectories which start on the face corresponding to e will remain in the intersection of R_e and B during the current time step. COHO now uses linearized, over approximation of the non-linear model:

$$F(x, t) \subseteq A \cdot x + b + \text{cube}(u) \quad (2.3)$$

where $\text{cube}(u) = \{z \mid \forall i. |z_i| \leq u_i\}$. The $\text{cube}(u)$ term allows the nonlinear relation to be approximated by a linear one plus an error term. This error



$$\dot{x} \in F(x)$$

$$F(x) \subseteq Ax + b + \mathbf{cube}(u), \forall x \in e_{bloat}$$

Figure 2.2: A Time Step of COHO

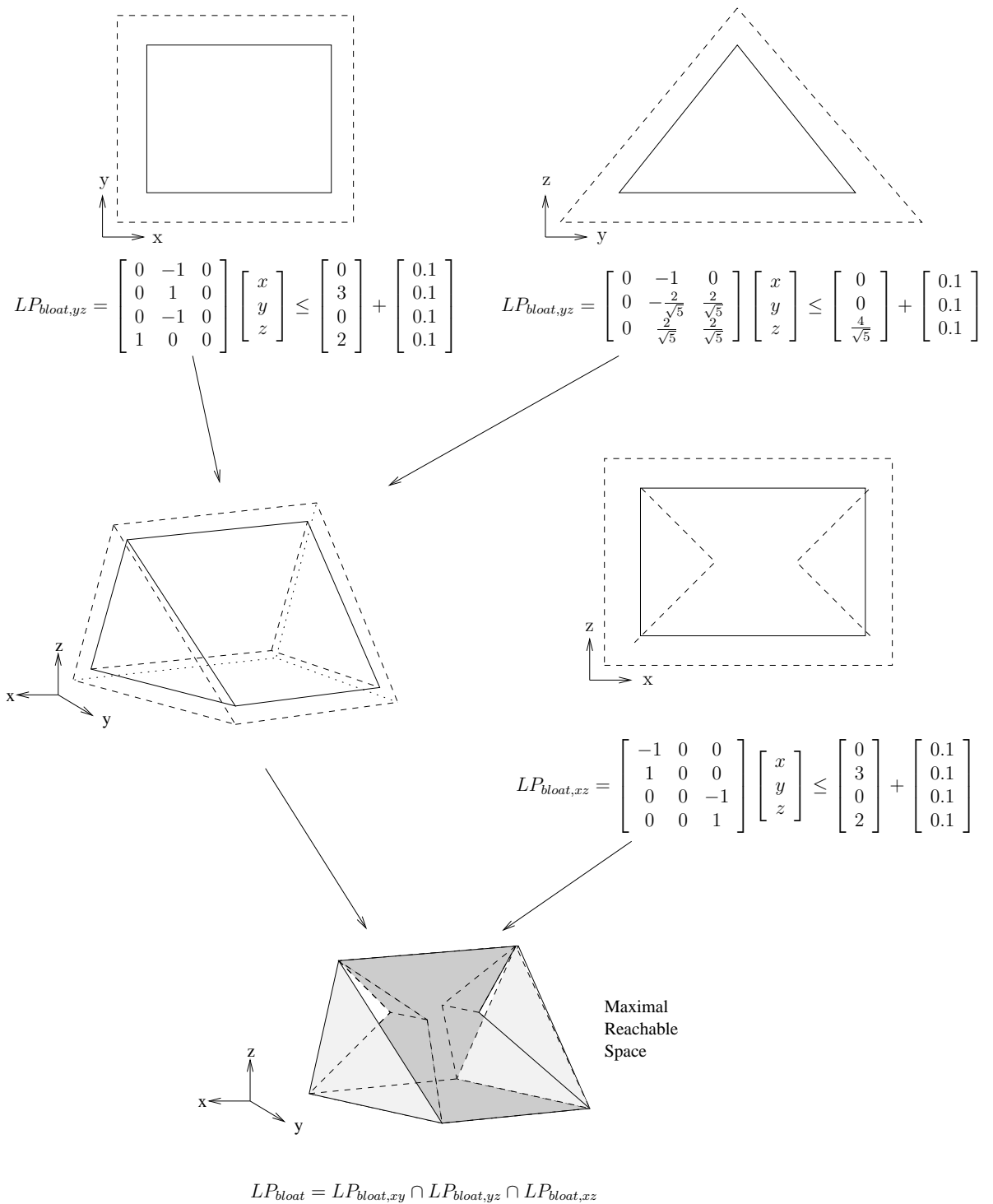


Figure 2.3: The Creation of a Bloated Linear Program from Projectagon [Laz01, Figure 2.2]

term is usually very small because the variables are restricted to the small rectangle R ; thus, the nonlinear ODE can be approximated by linear one with a small error bound. If the edge is too long to obtain a good approximation, COHO splits it into several short edges.

The linearization of the model is performed by code provided by the user. COHO calls a user provided Matlab function, `createmodel`, passing it the linear program for the intersection of R_e and B . The `createmodel` function can invoke the linear program solver to obtain bounds on the variables (and their linear combinations). Based on these bounds, `createmodel` computes values for the matrix A , and vectors b and u for equation 2.3. These are returned to COHO for further use in computing the time step. If the user provides a `createmodel` function that is incorrect, then COHO can produce invalid results.

3. Compute the step size

For each face, given the linearized model and the bloating distance, d_{bloat} , COHO can determine a step size such that all trajectories starting from this face are guaranteed to stay inside the region $B \cap R_e$ during the current time step. The next step size Δt is the minimum value of all such step size for each face, and the next step time t_{next} equals $t_{curr} + \Delta t$. Therefore, the reachable space at the end of the time step must be contained in $B \cap R_e$.

4. Advance each slice

Given the linearized model from step 2 and the step size from step 3, COHO computes the time-advanced slice with the following operations:

(a) Advance each face

COHO advances each slice by advancing each edge of the projection polygon for that edge (i.e. each face of the projectagon). Because the orientation of the face may rotate during the time step, its projection at the end of the time step can be a polygon, rather than an edge. Thus, we

refer to the post-image of an edge as an *edge polygon*. We first explain how COHO computes the edge polygons. We then describe how COHO combines these edge polygons to obtain a time advanced slice.

i. **Bloat the edge**

Recall that for each face, there is a corresponding edge of a projection polygon. Let e be such an edge. COHO constructs the rectangle R_e as described in step 2.

ii. **Intersect the bloated edge with original polygon**

The intersection of R_e and the original polygon for this slice is computed. The result polygon contains the current face and a small inward region. It might be non-convex, in which case, its convex hull is computed and used in the following steps.

iii. **Advance the intersection according to the linearized model**

The intersection from the previous section can be described by the following constraints:

$$Mx_0 \geq q \tag{2.4}$$

For simplicity, we first explain how the edge is advanced neglecting the approximation error, $\text{cube}(u)$. We then describe how COHO incorporates this error term. With this simplification, the linear model from step 2 becomes:

$$\dot{x} = Ax + b \tag{2.5}$$

The advanced region can be computed by integrating. Let x_e be the position at time t_{next} of a point whose position is x_0 at time t_{curr} . By integrating equation 2.5 for the time period $[t_{curr}, t_{next}]$, we obtain:

$$x_e = e^{A\Delta t}x_0 + (e^{A\Delta t} - I)A^{-1}b \tag{2.6}$$

The combination of equation 2.4 and equation 2.6 produces the con-

straints for the advanced face $F_{advance}$.

$$MEx_e \geq q_e \tag{2.7}$$

where $E = e^{-A\Delta t}$ and $q_e = q + M(I - e^{-A\Delta t})A^{-1} \Delta t$.

Of course, the actual model is linear differential inclusion, error bound must be considered for the advanced face. COHO does this by treating the error term as an input to the system. At any time instant, the worst-case such input will correspond to one of the corners of the error cube. Although the worst-case corner may change over the course of a time step, COHO makes the simplifying assumption that time steps are small, and that the same corner will be the worst-case one throughout the time step. With this assumption, COHO treats each dimension of the error cube separately to approximate the worst-case error due to non-linearity in the model. This is the one place where COHO could potentially compute a slight under approximation of the reachable space. This is an issue that has been present since the original design of COHO [GM99]. Addressing this potential unsoundness is beyond the scope of the present thesis and is a topic for future work.

iv. **Project back the advanced face onto the slice**

The feasible set for $F_{advance}$ is a convex polyhedron. As described in Chapter 5, COHO computes the projection of this polyhedron onto the two-dimensional subspace of this slice.

v. **Intersect the projection polygon with the bloated slice**

We know that the advanced face should be contained within the bloated region. Therefore, the intersection of the projection polygon and the bloated polygon is computed to get a tighter approximation.

(b) **Construct the advanced slice**

When the time-forward images of all of the edges of the slice have been

computed, COHO takes the union of these "edge polygons" to determine the boundary of the polygon at the end of the time step. Typically, this increases the number of edges in the polygon. Then, COHO performs overapproximating simplifications to keep the number of edges in the polygon tractable.

5. Clip advanced slices and create new projectagon

Using the steps described above, COHO produces a new polygon for each slice of the projectagon. These polygons define a projectagon that contains the reachable space at the end of the timestep. However, these polygons might not be feasible for each other. Therefore, COHO clips them (project the projectahedron onto each two dimensional subspace) to ensure that they are mutually feasible. The projectagon P_{next} of the next time step is created from these clipped polygons.

As seen from this short description, COHO makes extensive use of linear programs when computing reachable sets for systems modeled by non-linear ODEs. For example, step 4(a)ii, 4(a)iv, 4(a)v and 5 need the function to compute projection of a projectagon onto a two dimensional subspace. The projection function and step 3 use lots of linear programs.

When COHO was first implemented [GM99], Greenstreet and Mitchell found that the linear programs that arise in the reachability computation are often highly ill-conditioned. While they were able to analyse a few simple systems, this ill-conditioning prevented further application of the approach. More recently [Laz01], the special structure of the linear programs arising in COHO was used to obtain efficient and robust solutions. However, the error was still too large for badly conditioned problems, and the error bounds were not made available to the projection algorithm. Thus, the projection step often failed when applied to an ill-conditioned linear program.

These problems motivate us to apply interval arithmetic to the linear pro-

gram solver and design new algorithm for ill-condition system handling and projection computation.

Chapter 3

Architecture of the Coho System

The previous chapter included a description of prior work on the COHO verification tool that is the subject of this thesis. This chapter describes the organization of the COHO software with an emphasis on the packages that were developed or modified in the current research.

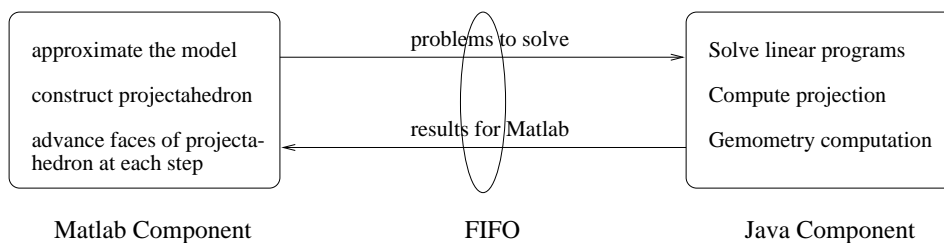


Figure 3.1: The Architecture of COHO System

At the top-level, COHO has two main components: a component written in Matlab and a component written in Java, as shown in Figure 3.1. The Matlab component provides the user interface. In particular, the model for the system being verified is written as a Matlab module. The user interacts with COHO through a set of Matlab functions. These functions support creating a model, defining projectahe-

dra and computing reachable regions. The Java component provides functionality that is not readily available in Matlab. In particular, it provides operations for manipulating the projection polygons and the robust linear program solver that is the focus of this thesis.

The Java and Matlab components communicate through a pair of named pipes. There is a C program invoked by Matlab process that creates these pipes. Options provided on the command line of the C program allow the content of the streams to be logged. Furthermore, the communication between the two programs uses plain ASCII hence is human readable. This capability facilitates developing and debugging the Java and Matlab functions separately. To use the functions from the Java component, the Matlab component writes an expression on the `m2j` (Matlab-to-Java) pipe. The Java component includes a simple interpreter that evaluates these expressions and writes the results back on the `j2m` (Java-to-Matlab) pipe. Matlab and Java programs can send numerical values either using scientific notation or as a hex encoding of the bits. The later makes the files harder to read but ensures absolutely no loss of precision.

3.1 Java Component

Figure 3.2 shows the top level structure of the Java component. Our robust linear program solver is implemented in the `lp` packages and uses classes from the `number` and `matrix` packages.

The Java component consists of four subsystems: an interpreter for matlab-java interface, geometry computation packages, infrastructure packages, and our linear programming packages. The interpreter employs JLex [BA] and CUP [SHA] for lexical analysis and parsing. It interprets commands, from the `m2j` (Matlab-to-Java) pipe and calls the corresponding functions from other packages. The old version interpreter is used with only minor modification and it is not described further here. The geometry package is used for polygon operation, after each advance step.

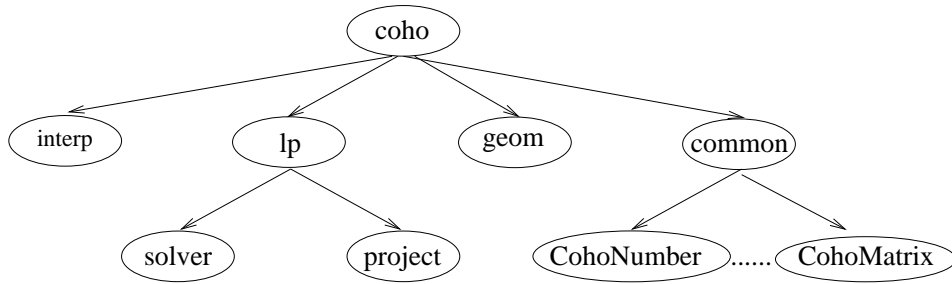


Figure 3.2: Package Hierarchy of COHO System

It make extensive use of the Bentley-Ottman algorithm [BO79] [PS85, 271-277]. The number and matrix packages are described in the following sections. The linear program solver is presented in Chapter 4. The linear program projection is described in Chapter 5.

3.2 Number Package

To implement the linear program solver, interval arithmetic is employed to track computation error. Because round off error is inevitable for floating point computation, the error also accumulates and no error bound is provided. For COHO verification system, under-approximation is unacceptable; on the other hand, interval computation is reliable because it produces both results and error bounds, which allow us to guarantee over approximation.

A new interface is required to implement interval number. Interval number can not extend the build-in data type `java.lang.Double` (also `Integer`, `Boolean`, etc) because it is a final class. Furthermore, `java.lang.Number` does not provide the arithmetic operations that we need to define standard matrix operations and implement our linear programming algorithm. The new interface is called `CohoNumber` rather than `Number` to avoid name confusion. `CohoNumber` extends `java.lang.Comparable`.

Several implementations of `CohoNumber` are provided in the number package. The inheritance structure of these classes is show in Figure 3.3. `CohoNumber`

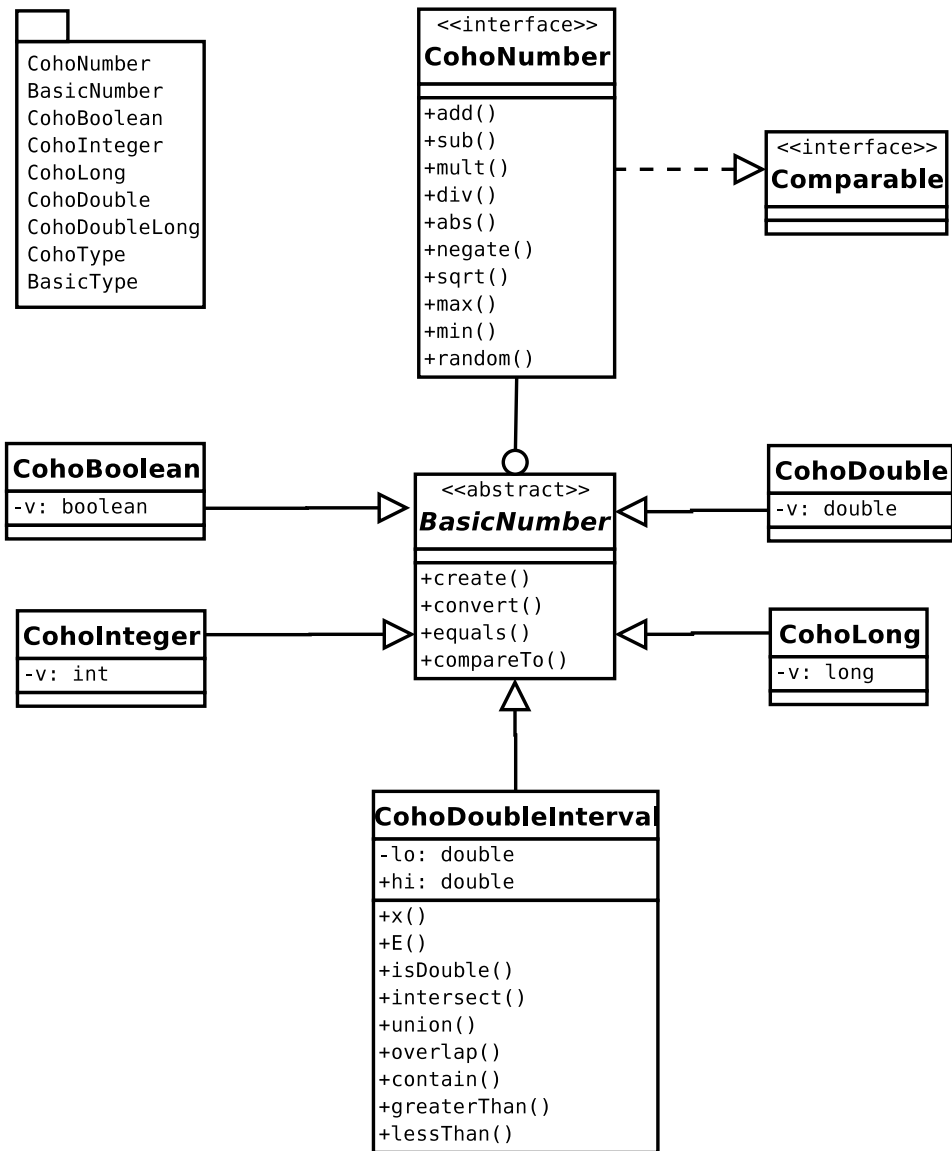


Figure 3.3: Class Hierarchy of Number Package

provides functions for standard arithmetic, including *add*, *sub*, *mult*, *div*, *abs*, *max*, *min*, etc. This allows other classes to build on `CohoNumber` and work with any underlying representation of the data.

The `BasicNumber` class defines an abstract number that implements the `CohoNumber` interface. It allows a new implementation of `CohoNumber` to provide a minimal set of arithmetic and comparison operations. Promotion is performed automatically when a binary operator is applied to different kinds of `CohoNumbers` for extensibility. The order of promotion, defined in the `BasicType` class, is *CohoBoolean* \rightarrow *CohoInteger* \rightarrow *CohoLong* \rightarrow *CohoDouble* \rightarrow *CohoDoubleInterval*, with possible loss of precision when promoting a `CohoLong` to a `CohoDouble` for longer number n with $|n| > 2^{53}$. If a new `CohoNumber` is added, we do not need to modify the `BasicNumber` or other `CohoNumbers` to provide the appropriate promotions. After defining the promotion sequence in the `BasicType` class, the implementation is independent.

Our main concern is `CohoDoubleInterval`, the class for interval numbers. `CohoDoubleInterval` is a representation of number x by a range from its lower bound to its upper bound. In the following, we use boldface like \mathbf{x} for interval values, and \bar{x} , \underline{x} for its upper and lower bound.

`CohoDoubleInterval` implements the four basic operations as:

$$\begin{aligned} \mathbf{x} + \mathbf{y} &= [\underline{x} + \underline{y}, \bar{x} + \bar{y}] \\ \mathbf{x} - \mathbf{y} &= [\underline{x} - \bar{y}, \bar{x} - \underline{y}] \\ \mathbf{x} \times \mathbf{y} &= [\min(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}), \max(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y})] \\ 1/\mathbf{x} &= [1/\bar{x}, 1/\underline{x}] \text{ if } \underline{x}\bar{x} > 0 \end{aligned}$$

Obviously, these functions satisfy the *inclusion property* [DMn05]:

$$\mathbf{x} \otimes \mathbf{y} = \{x \otimes y | x \in \mathbf{x}, y \in \mathbf{y}\}, \text{ for } \otimes \in \{+, -, \times, /\} \quad (3.1)$$

However, for general interval arithmetic, the lower (upper) bound can not be represented exactly in computer. Therefore, round off error must be considered for

interval computation. Our implementation uses the property that the IEEE floating point standard requires rounding to the nearest representable number. Thus, the double precision result is within $\frac{1}{2}ulp$ (unit of least precision) of the exact value. However, the smallest value to be added is a single ulp ; thus, the interval is widened by $2ulp$ during each operation.

Several other operations are also implemented as:

$$\begin{aligned}
-\mathbf{x} &= [-\bar{x}, -\underline{x}] \\
|\mathbf{x}| &= \begin{cases} [\min(|\underline{x}|, |\bar{x}|), \max(|\underline{x}|, |\bar{x}|)] & \text{if } \underline{x}\bar{x} > 0 \\ [0, \max(|\underline{x}|, |\bar{x}|)] & \text{if } \underline{x}\bar{x} \leq 0 \end{cases} \\
\max(\mathbf{x}, \mathbf{y}) &= [\max(\underline{x}, \underline{y}), \max(\bar{x}, \bar{y})] \\
\min(\mathbf{x}, \mathbf{y}) &= [\min(\underline{x}, \underline{y}), \min(\bar{x}, \bar{y})] \\
\mathbf{x} \cup \mathbf{y} &= [\min(\underline{x}, \underline{y}), \max(\bar{x}, \bar{y})] \\
\mathbf{x} \cap \mathbf{y} &= \begin{cases} [\max(\underline{x}, \underline{y}), \min(\bar{x}, \bar{y})] & \text{if } \max(\underline{x}, \underline{y}) \leq \min(\bar{x}, \bar{y}) \\ \text{exception} & \text{otherwise} \end{cases} \\
\sqrt{\mathbf{x}} &= \begin{cases} [\sqrt{\underline{x}}, \sqrt{\bar{x}}] & \text{if } \underline{x} \geq 0 \\ \text{exception} & \text{otherwise} \end{cases}
\end{aligned}$$

Comparison operations for interval are different because the result can be ambiguous. If two intervals overlap, the `compareTo` function throws a `NotComparableIntervalException`. Additional comparison operations are also provided, including `greaterThan`, `lessThan`, `geq`, `leq`, `neq` etc. Table 3.1 shows the details.

The IEEE floating point standard [Gol91] defines rounding modes including round-up and round-down. Thus, a hand-coded implementation that used the full capability of the hardware would probably be faster and provide somewhat tighter error bounds. This will be described in greater detail in Chapter 7.

Table 3.1: Functions of CohoDoubleInterval

Function	Description	Example
greaterThan	return true if x is clearly greater than y; otherwise, return false .	[2,3] is greater than [0,1] but not greater than [1,2]
lessThan	return true if x is clearly less than y; return false otherwise.	[2,3] is less than [4,5] but not less than [3,4]
geq	greater than or equal	[2,3] is greater than or equal [1,2]
leq	less than or equal	[1,2] is less than or equal [2,3]
neq	not the same double value	[1,2] is not equal with [1,2]
same	return true if x and y have the same lower and upper bound	[1,2] is the same with [1,2]
compareTo	return 1 if x is clearly greather than y, return -1 if x is clearly less than y; return 0 if x and y are the same double value, throw exception otherwise	[1,3] compareTo [2,4] throws an exception
equal	return true if x and y are the same double value, return false otherwise	[1,2] is not equal with [1,2], [1,1] equals [1,1]
overlap	return true if the intersection of x and y is not empty	[1,2] and [2,3] overlaps
contain	return true if x contains y	[2,4] contains[3,4]

3.3 Matrix Package

The matrix package provides matrix computation based on the number package. The elements of CohoMatrix are of instances of CohoNumber. This package includes a Matrix interface and several kinds of matrices. The package structure is shown in Figure 3.4. Basic matrix operations are provided, such as get or assign value; arithmetic operations including transpose, add, sub, mult, etc; operation for rows (columns) and comparison operations.

BasicMatrix implements most functions that are independent of the specific element data type. Mapper is used to implement elementwise operations, such as the addition of two matrices. Reduce is for operations that compute a single result from all of the elements, such as the maximum element of a matrix. Using this capability, very few methods of BasicMatrix are abstract. In practice, a concrete subclass of

BasicMatrix only need to provide methods that assign a value to an element, get the value of an element, and create an object of the element type. This makes it very easy to create new version of Matrix. Of course, the subclass can override some of the default implementation of methods for efficiency. For example, CohoMatrix overrides the create function because we do not need to create the zero elements for a CohoMatrix.

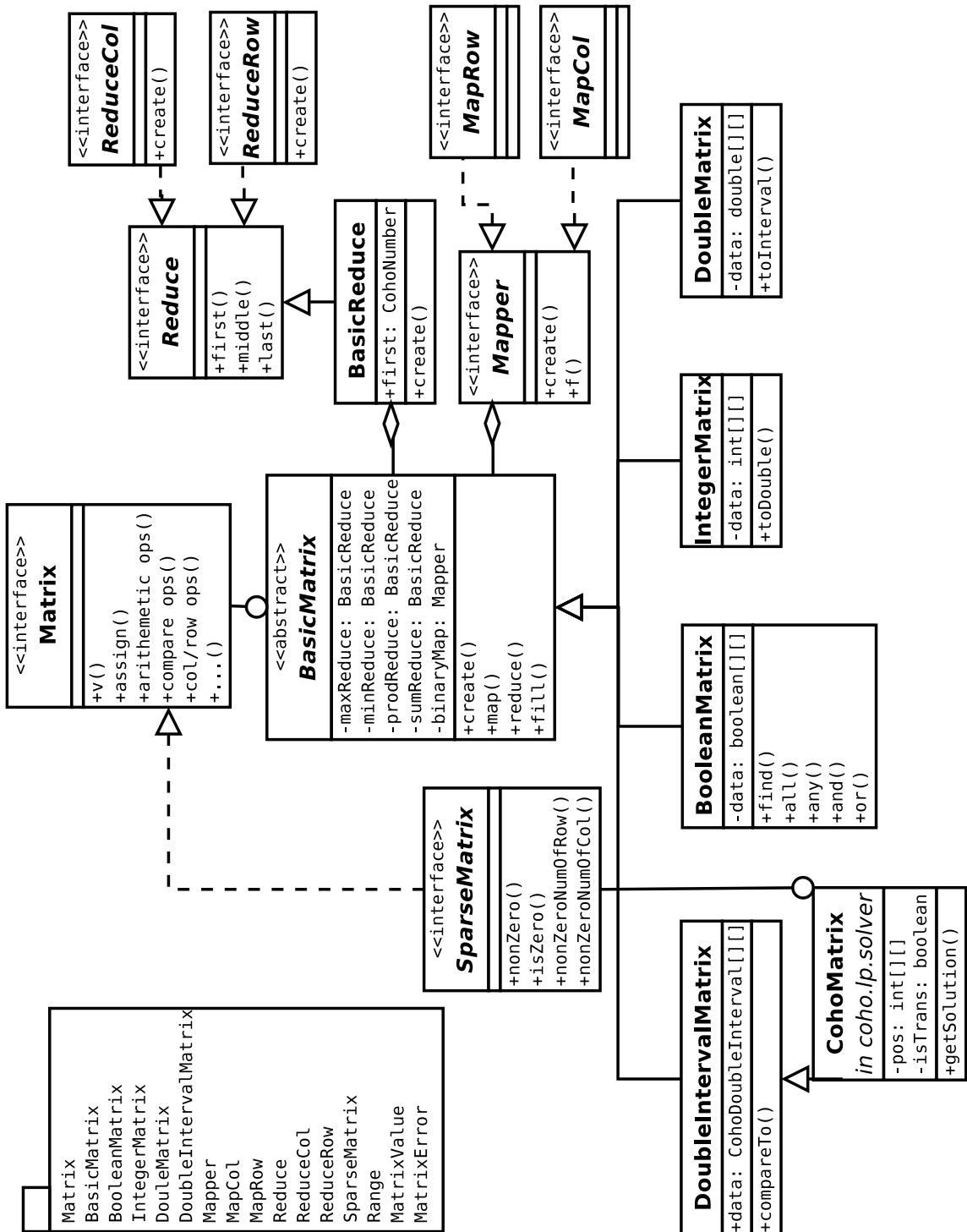


Figure 3.4: Class Hierarchy of Matrix Package

Chapter 4

Coho Linear Program Solver

As described in Chapter 2, linear programming plays an important role in COHO. All linear programs are of a special form that we refer to as a COHO *Linear Program*. We implemented an efficient and reliable solver based on the Simplex algorithm [Dan63] for it.

Laza has set up the framework for COHO solver. What has been accomplished by Laza includes:

1. Algorithm and implementation of an $O(n)$ linear system solver for COHO matrix [Laz01, pp.32-36].
2. Algorithm to deal with uncertainty and avoidance of cycling [Laz01, pp.87-90].
3. Error analysis of COHO solver [Laz01, pp39-81].

We extend Laza's algorithm and reimplemented the solver using interval arithmetic. The contribution includes:

1. Redesign the architecture of COHO solver.
2. Implement the solver based on interval numbers, using the *Number* and *Matrix* package described in Chapter 3.2,3.3. New solver interface and pivot rule are proposed.

3. Extends the linear system solver for both COHO matrix and COHO dual matrix, implements a hybrid solver to reduce error.
4. Dealing with highly ill-conditioned linear programs; a new method to estimate the condition number is presented.

This chapter is structured as follows: Linear programming and Simplex algorithm is introduced in the first section. Section 4.2 describes solutions for problems caused by interval arithmetic. Laza's linear system solver is presented in section 4.3. COHO linear programs can be badly conditioned, solution is presented in section 4.4. At the last section, implementation issues are described.

4.1 Linear Programming and the Simplex Algorithm

4.1.1 Coho Linear Programming

A Linear Programming (LP) problem is an optimization problem in which the objective function and constraints are all linear.

A linear program of standard form is

$$\begin{aligned}
 \min c^T x \quad & s.t. \\
 Ax = b & \\
 x \geq 0 &
 \end{aligned} \tag{4.1}$$

The column matrix c is called the *cost vector* or *optimization direction*, while the value x_{opt} that minimizes $c^T x$ is the *optimal vertex* or *optimal point*, and $c^T x_{opt}$ is the *optimal value* or *optimal cost* for the linear program. The constraints define the *feasible region* of the linear program. Suppose there are n variables and m equality constraints, then the feasible region for standard linear program is the non-negative portion of a polyhedron with m faces in n -dimensional space.

A set of m linearly independent columns of matrix A is called *basis*, denoted as \mathcal{B} . The columns in the basis are called *basic columns* whereas others are *non-basic*. Basic columns corresponds to *basic variables* with value of $t_0 = A_{\mathcal{B}}^{-1}b$. The

basic solution for a basis \mathcal{B} is defined as

$$x_j = \begin{cases} 0 & \text{if } j \notin \mathcal{B} \\ t_{0,k} & \text{if } j = \mathcal{B}_k \end{cases} \quad (4.2)$$

The linear programs in COHO express a conjunction of the half-spaces corresponding to the projection polygon edges. These linear programs can be written

$$\begin{aligned} \min c^T x & \quad s.t. \\ P \cdot E^{-1}x & \geq b \end{aligned} \quad (4.3)$$

where $P \cdot x \geq b$ are the constraints such that x is a point of the projectagon defined in Chapter 2; c is the cost vector; and E is the forward Euler operator for the linearized model. Rows of P correspond to projection polygon edges; therefore, each row has either one or two non-zero elements. Because the Euler steps are relatively small, E is well-conditioned, and E^{-1} is easily computed. We refer to a linear program in the form of equation 4.3 as a COHO *linear program*. And P is called a COHO *matrix* defined as the matrix with one or two nonzero elements in each row.

4.1.2 Primal-Dual Method

COHO linear programs can obviously be reduced to standard form by adding extra variables. However, this will destroy the special structure of original COHO linear program. We solve the problem using its dual form that retains the special structure. The dual [PS82] of a COHO LP is a standard form linear program. We write the COHO *dual linear program* or *standard form COHO LP* as

$$\begin{aligned} \min -b^T u & \quad s.t. \\ P^T u & = E^T c \\ u & \geq 0 \end{aligned} \quad (4.4)$$

and P^T has one or two nonzero elements in each column which is called a COHO *dual matrix*.

The primal/dual pair linear programs have many useful properties. By the *Duality Theorem* [Van01, pp.58-67], the primal and dual linear programs have the same optimal basis and optimal value. Any non-optimal feasible basis for the dual problem is infeasible for the primal, and produces an upper bound for the primal problem, and vice versa. The optimal basis is the one that is feasible for both primal and dual problems. Therefore, the COHO LP can be solved by applying Simplex to the standard form COHO LP as described in the following.

4.1.3 Simplex Algorithm

There is a classical algorithm for linear programming: Simplex. However, it only operates on standard form linear programs. Therefore, the COHO linear programs are converted into COHO dual linear programs first, then solved by our modified Simplex solver. In addition, an initial feasible basis must be provided to Simplex. Finding a feasible basis is not trivial, and it will be dealt with in Section 4.2.

Simplex is a greedy algorithm. During each step, it tries to replace one column of the current basis with a new column to obtain another feasible basis that lowers the cost. Let \mathcal{B} be the current basis, $A_{\mathcal{B}}$ denotes A restricted to this basis. Let t_j be the column vector defined by:

$$t_j = A_{\mathcal{B}}^{-1} A_{:,j} \quad (4.5)$$

where $A_{:,j}$ is the j^{th} column of matrix A . The variable

$$\bar{c}_j = c_j - c_{\mathcal{B}}^T t_j \quad (4.6)$$

is the *relative cost* of the j^{th} column with respect to current basis \mathcal{B} . If the relative cost is negative, the cost is reduced by introducing the j^{th} column into the basis and evicting some basic column.

The algorithm must determine which column in the basis to eject when a new column enters. The decision is guided by the requirement that the new basis

must be feasible and the cost is reduced. The index of column to be evicted is

$$k = \arg \min_i \frac{t_{0,j}}{t_{j,i}} \quad (4.7)$$

The action of replacing a column in the old basis with a new column to get a more favorable basis is called *pivoting*. Simplex pivots to reduce cost until the optimal basis is found.

The cost reduction for the pivot is

$$cost_{reduce} = \frac{t_{0,j}}{t_{k,j}} \cdot \bar{c}_j \quad (4.8)$$

To reduce the cost, $cost_{reduce}$ must be negative, therefore $\frac{t_{0,j}}{t_{k,j}}$ should be positive. Because $P_{\mathcal{B}}$ is feasible basis, which implies $t_{0,j}$ is non-negative, then $t_{k,j}$ should be negative for a success pivoting. If $t_{0,j}$ is zero, there is no cost reduction, and the bases before and after such a pivot are called *degenerate*. Furthermore, if $t_{j,i} < 0, \forall i = 1, \dots, n$, then the linear program is *unbounded*, feasible points with arbitrary low cost exist. COHO linear program can have degeneracy, but it can not be unbounded because its feasible region is an bounded polyhedron.

In the absence of degeneracy, Simplex works because at each pivot it reduces the cost while there are finite number of feasible bases. In the presence of degeneracy, Simplex can loop indefinitely through a set of degenerate bases, which is called *cycling*. Cycling avoidance is achieved by Bland's anticycling algorithm [Bla77],[PS82]. Simplex can not guarantees finding the shortest path from the initial feasible basis to the optimal basis, and in fact it's an exponential algorithm in the worst case. However, it performs quite well in practice.

4.2 Combination of Simplex and Interval Computation

The Simplex algorithm described above assumes that all operations are free of errors. Unfortunately, floating point computation introduces round off error. Errors in the computation may prevent Simplex from producing the correct result. Although the

accumulated error is reduced largely by Laza’s efficient linear system solver [Laz01, pp.32-36], it can not be eliminated.

For example, Simplex depends on comparisons between floating-point numbers to find the new basic column and evicted column. In the presence of ill-conditioning, the result may be incorrect. Clearly, a wrong pivoting can make the algorithm fail:

- If a wrong basic column is evicted out of the basis, an infeasible basis is reached.
- If a wrong column is brought into the basis, a more costly basis is reached that make the algorithm fall into a cycle of bases.

At the same time, COHO requires an over-approximation result for verification. The optimal value from Simplex may be under-approximated because of nearest representation rounding off, even though the optimal basis is correct.

To solve all problems above, we combine the Simplex algorithm and interval computation. All values used in the solver are interval numbers rather than floating point values, as described in Section 3.2. The interval based Simplex algorithm returns a set of possible optimal bases and a range that contains the optimal value. Therefore, over-approximation is guaranteed, and an error bound is provided. In addition, the ill-condition problems are easily detected using interval arithmetic as described in Section 4.4.

However, using interval arithmetic, the outcome of comparisons of interval numbers can be ambiguous: if the intervals for x and y overlap, then the ordering of x and y cannot be determined. Therefore, the introduction of interval representation introduces several problems:

1. Pivots *might* be favourable or not.
2. Bases encountered *might* be feasible and/or optimal.

3. The algorithm for finding the initial feasible basis proposed by Laza is no longer guaranteed to succeed.

The following describes our interval based solver and algorithm to handle these problems. These methods are not restricted to COHO linear programs, it modifies the Simplex algorithm thus works for general linear programs.

4.2.1 Ambiguous Pivoting and Cycling

The Simplex algorithm works by pivoting to bases that progressively lower the cost until the optimal basis is reached. These pivots are determined from equations 4.6 and 4.7. However, interval arithmetic operations in our solver make it possible that there are several ambiguous pivots and no clearly favorable choice. For example, the relative cost may not be clearly positive or negative. Thus, it might be impossible to find a column to bring in such that the new basis is clearly favorable. Similarly, it might be impossible to find the basic column to evict. Therefore, classical Simplex algorithm does not work if there is no clearly favorable bases because of interval computation.

To deal with pivot selection, we use Laza's method [Laz01, pp.87-90]. The algorithm first tries to find a clearly favourable pivot if possible. If uncertainty happens during interval comparisons, it tries all *possibly* favorable pivots.

In our implementation, there are two steps of a pivot. Firstly, the algorithm finds the column to bring in, determined by relative cost from equation 4.6. It first checks to see if there is a column whose relative cost is clearly negative. This involves checking each column that is outside of the current basis until one with clearly negative relative cost is found. Each such check requires solving one linear system in linear time, which will be described in the next section. Thus, the total time is $O(n(m - n))$ where n is the number of constraints and m is the number of variables. If such column is found, it is taken, and the other columns do not need to be evaluated at this step. If not, then all the columns with possibly negative

relative cost are considered. In the second step, the algorithm find the basic column to evict, using equation 4.7. If there is a clear result from the *argmin* function, then it is taken; otherwise, all basic columns that might be evicted are considered. At this point, we have a set of possibly favorable pivots, our algorithm branches: all are explored, as shown in Figure 4.1.

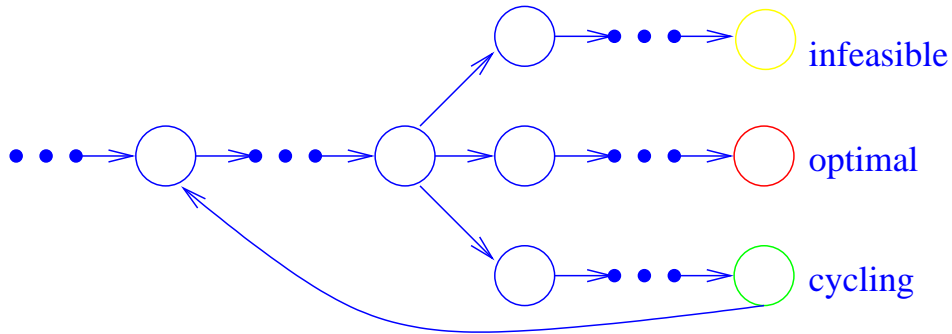


Figure 4.1: Pivot Selection of COHO Linear Program Solver. Branching leads to three types of bases: 1)infeasible basis; 2) optimal basis as expected; 3)more costly basis which causes a loop.

This branching pivoting may lead to a more costly basis thus make the algorithm fall into a cycle, which precludes the use of standard anti-cycling algorithms [Bla77]. Instead, we maintain a hashtable of all bases that we have visited. If a pivot leads to a basis that we have already seen, it is ignored. It is also possible that infeasible bases are encountered. If a basis is clearly infeasible, then it is removed by the algorithm without changing the optimal result. Otherwise, it must be quite close to the feasible region; therefore the error introduced is very small. By branching, COHO is guaranteed to visit the truly optimal basis.

The branching pivoting may lead to an exponential algorithm in theory. However, branching is rare in practice. Thus, the algorithm remains efficient.

4.2.2 Possible Optimal Result

Similar issues arise when testing bases for optimality. Mathematically, a basis is optimal if and only if it is feasible for both the standard- and COHO-form LPs. If COHO encounters a clearly optimal basis, it is done. Otherwise, each time COHO encounters a basis that is possibly optimal, it checks to see if the corresponding basis for the COHO-form LP is possibly feasible. If so, the basis is flagged as possibly optimal. If no clearly optimal basis is found, COHO uses the set of possibly optimal bases to determine an interval that it guaranteed to contain the actual cost at optimality.

Therefore, we redefine the interface for the linear program solver. The result it returns is a bound for its optimal cost, which is usually very small, and a series of possible optimal bases, which definitely contains the real optimal basis.

4.2.3 Initial Feasible Basis

Simplex requires an initial feasible basis, while finding a feasible basis is non-trivial for general LP. Laza [Laz01, 82-89] presented a simple method to find a feasible basis. Laza's algorithm finds an invertible basis first, then constructs a helper LP: the invertible basis is feasible for the helper LP, and the optimal basis of the helper LP is feasible for COHO LP.

Laza's approach does not work when interval arithmetic is used. As described above, the solver can only return a numerically *possible* optimal basis for the helper LP thus a *possible* feasible basis for the original COHO LP. If the basis is actually infeasible, then this algorithm fails.

Rather, we apply the *Big M method* [Win87]. For a COHO *Dual LP*:

$$\begin{array}{l}
 \min(c_1 y_1 + \dots + c_n y_n) \\
 s.t. \quad \begin{bmatrix} a_{11} & \cdot & a_{1n} \\ \cdot & \cdot & \cdot \\ a_{m1} & \cdot & a_{mn} \end{bmatrix} \begin{bmatrix} y_1 \\ \cdot \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \cdot \\ b_m \end{bmatrix}
 \end{array}$$

$$y_i \geq 0 \quad i = 1, \dots, n$$

we add m *extra variables* z_1, \dots, z_m , and make them expensive by assigning this a very large cost, M ; thus the extra variables must be driven out of the basis by the normal pivoting procedure of Simplex. We append an identity matrix to the end of COHO *Dual Matrix* A ¹:

$$\begin{aligned} & \min(c_1 y_1 + \dots + c_n y_n + M z_1 + \dots + M z_m) \\ \text{s.t.} \quad & \begin{bmatrix} a_{11} & \cdot & a_{1n} & 1 & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m1} & \cdot & a_{mn} & 0 & \cdot & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ \cdot \\ y_n \\ z_1 \\ \cdot \\ z_m \end{bmatrix} = \begin{bmatrix} b_1 \\ \cdot \\ b_m \end{bmatrix} \\ & y_i \geq 0 \quad i = 1, \dots, n \quad z_j \geq 0 \quad j = 1, \dots, m \end{aligned}$$

We can see this is also a COHO *Dual LP*, it has the same optimal value and optimal basis as the original one because the added variables z_1, \dots, z_m are expensive and will not appear in the optimal basis. This is equivalent with adding some redundant constraints, similar with $x_i \leq M$, to the COHO *LP*. Clearly, the added *extra columns* are a *clearly feasible basis* for this helper LP. The *extra column* can not appear in the optimal basis. However, there are LPs for which we can not drive these columns out no matter how big M is. For example, when some $b_i = 0$, then $z_i = 0$, the cost for it is zero and thus can not be driven out. So we should not introduce such columns during pivoting step; and even if the *relative cost* of column j is zero, we should try to bring it into basis, because although it can not reduce the cost, it may drive out the *extra columns* with the same cost. At last, we should remove any possibly optimal bases which contain such undesirable columns.

¹Here we assume b_i are all nonnegative. If b_i is negative, we just set the i^{th} diagonal element as -1 to make the basis feasible

4.3 Efficient Linear System Solver

So far, we have developed an interval based linear program solver. It is not as efficient as the classical Simplex algorithm, although it has several advantages. In the algorithm, solving the linear systems to compute the tableau is critical for performance. Laza [Laz01, pp.32-36] has developed a linear system solver, taking advantage of the special structure of COHO matrix, that improves efficiency and reduces accumulated error. We reimplemented his algorithm using interval arithmetic and extended the algorithm for both COHO matrices and COHO dual matrices. We also implemented a hybrid algorithm that improves the accuracy of the result with a small time penalty.

Traditional implementations of Simplex exploit the property that each basis is a rank-1 update [GV96] of its predecessor. This allows the updated tableau to be computed in $O(n(n-m))$ time where n is the number of constraints in the standard form LP, and m is the number of variables. However, this approach accumulates error from one step to the next. Using interval arithmetic, the error bounds would quickly become larger than the values themselves. Instead, we exploit the structure of the P matrix to obtain a linear time algorithm for solving the linear systems that arise in the Simplex algorithm without error accumulation.

As noted earlier, each row of P corresponds to an edge of a projection polygon and thus has either one or two non-zero elements. Accordingly, each column of matrix $P_{\mathcal{B}}^T$ has either one or two non-zero elements. Now, consider the rows of $P_{\mathcal{B}}^T$. By the pivot selection rules, $P_{\mathcal{B}}^T$ will never have a row that is all zeros. If a row has exactly one non-zero element, then the value of the corresponding variable can be determined directly and the row and column are eliminated. Such elimination can be continued until every row has at least two non-zero elements. These eliminations preserve the property that every column has at most two non-zero elements, and the matrix remains square. Thus, at the end of this phase, every row and every column of the matrix has exactly two non-zero elements. This matrix has many wonderful

properties; so, we will call this matrix W .

Assume that W is $n \times n$. Consider an undirected graph with vertices $v_1 \dots v_n$ such that there is an edge between vertices i and j iff W has a column that has non-zero elements in rows i and j . By the structure of W , every vertex in this graph has degree 2, and the graph can be decomposed into disjoint, simple cycles. The linear systems corresponding to these cycles can be solved independently. For simplicity of presentation, we will assume that W has a single cycle and note that the generalization to multiple cycles is straightforward.

We can now permute the rows and columns of W such that $W_{i,j} \neq 0$ iff $j = i$ or $j = (i \bmod n) + 1$. We can scale the rows so that every diagonal element of the matrix is 1. The resulting matrix has the form shown below:

$$W = \begin{bmatrix} 1 & -\alpha_1 & 0 & \dots & 0 \\ 0 & 1 & -\alpha_2 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 1 & -\alpha_{n-1} \\ -\alpha_n & 0 & \dots & 0 & 1 \end{bmatrix} \quad (4.9)$$

Now, to solve a linear system of the form $Wu = d$, we observe that

$$u_1 = \frac{\sum_{j=1}^n (d_j \beta_{j-1})}{1 - \beta_n} \quad (4.10)$$

$$u_{i+1} = \frac{1 - \sum_{j=1}^i (d_j \beta_{j-1})}{\beta_i} \quad (4.11)$$

where $\beta_k = \prod_{i=1}^k \alpha_i$.

Figure 4.2 summarizes the flow of our linear system solver. It is straightforward to show that all of the steps described above for solving a linear system can be performed in $O(n)$ time where n is the number of equality constraints in the standard form LP. Furthermore, the only step that introduces cancellation is the denominator $1 - \beta_n$ in equation 4.10. Thus, we can rapidly identify ill-conditioned linear systems.

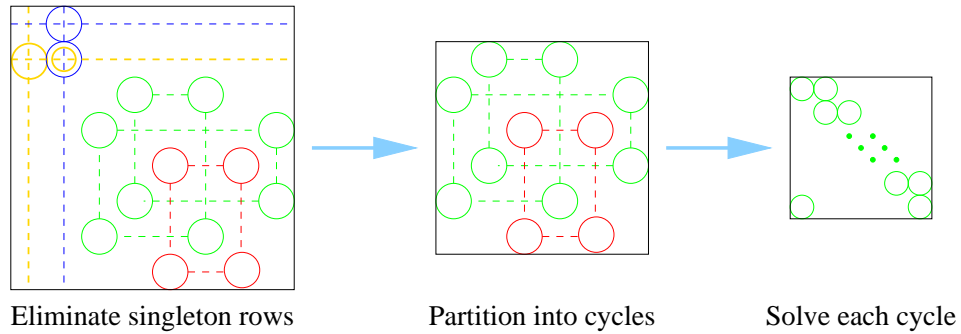


Figure 4.2: COHO Linear System Solver

In the event that a basis is poorly conditioned, we can use equation 4.10 to solve for each variable separately. This results in an $O(n^2)$ algorithm but avoids accumulation of error. In our implementation, we use the linear time method first, and if the error exceeds a preset bound, solve again with the more precise $O(n^2)$ algorithm. In practice, the $O(n)$ algorithm is used almost all of the time. Thus we obtain high accuracy and low run-time.

4.4 Ill-Conditioned Basis and Exception Handling

The solver described above works efficiently and reliably for well-conditioned COHO linear programs. However, when the problem is too badly conditioned, for example, when P_n is close to 1 in equation 4.10, the error bound of the linear system solver might be so large that the optimal value is overapproximated too much to be useful. In these cases, a slightly primal-infeasible but well-conditioned basis is a better overapproximation of the optimal cost.

4.4.1 Conditioning

Interval computation provides an error bound that makes it quite easy to detect badly conditioned problems. However, this method requires solving the linear systems even though it is badly conditioned. We need a fast method to detect badly

ill-conditioned problems.

The condition number of a matrix is the ratio of the largest to smallest singular value in the singular value decomposition, it is an estimate of how many digits are lost in solving a linear system with that matrix. But it is also expensive to compute the exact condition number. An approximation technique is required.

Laza proposed an algorithm to compute the upper bound on the condition number of COHO matrix [Laz01, pp.41-46]. However, the estimation is often much larger than the actual condition number. Thus, problems can be needlessly identified as badly conditioned at beginning. Furthermore, the estimation for a matrix and its transpose can be different, which makes it possible that a basis is well-conditioned for COHO dual linear program while ill-conditioned for COHO linear program. In the implementation of solver, unexpected exception might be thrown at the final step that checks the feasibility of bases for COHO linear programs. Finally, the computation of condition number should be much faster than the linear system solver, otherwise approximation is neither efficient nor necessary because interval arithmetic provides error bound. Laza's computation is not easy enough compared with his fast linear system solver.

We developed a new algorithm for computing lower bound of the condition number of a COHO matrix. It catches most badly conditioned problems. When a badly conditioned problem skips through, the intervals of the result are very large, and the problem is then detected.

We use the ratio of the largest to smallest eigenvalue of the matrix as the underapproximation. The eigenvalue for COHO matrix or COHO dual matrix is easy to compute. Let λ denotes the eigenvalues of COHO matrix or COHO dual matrix A , we have

$$(1 - \lambda)^n = \prod_{i=1}^n \alpha_i \tag{4.12}$$

therefore, the eigenvalues are

$$\lambda = 1 - \beta_n^{1/n} \left(\cos \frac{2k\pi}{n} + i \cdot \sin \frac{2k\pi}{n} \right) \tag{4.13}$$

and our estimation is

$$\frac{\max(|\lambda_i|)}{\min(|\lambda_i|)} = \begin{cases} \frac{1+P_n^{1/n}}{|1-P_n^{1/n}|} & \text{if } n \text{ is even} \\ \frac{\sqrt{1+P_n^{2/n}-2P_n^{1/n}\cos\frac{2\pi\lceil n/2\rceil}{n}}}{|1-P_n^{1/n}|} & \text{if } n \text{ is odd} \end{cases} \quad (4.14)$$

Now, let us prove it is the lower bound of condition number.

Theorem 4.4.1 *For a nonsingular matrix A , let $\lambda_1, \dots, \lambda_n$ be the eigenvalues of A and $\sigma_1, \dots, \sigma_n$ be the singular values, then $\frac{\max(|\lambda_i|)}{\min(|\lambda_i|)} \leq \frac{\max(\sigma_i)}{\min(\sigma_i)}$*

Proof Suppose λ is one eigenvalue of A , then \exists an eigenvector $x \neq 0$ such that $Ax = \lambda x$. By the property of norm

$$\lambda \|x\| = \|\lambda\| \cdot \|x\| = \|Ax\| \leq \|A\| \cdot \|x\| \quad (4.15)$$

therefore,

$$\lambda \leq \|A\| \quad (4.16)$$

and it is straightforward that

$$\max(|\lambda_i|) \leq \|A\| \quad (4.17)$$

Similarly, for matrix A^{-1} , have

$$\max(|\lambda_i^{-1}|) \leq \|A^{-1}\| \quad (4.18)$$

and because eigenvalues of A^{-1} are reciprocal of that of A , thus

$$\begin{aligned} \|A\| \cdot \|A^{-1}\| &\geq \max(|\lambda_i|) \cdot \max(|\lambda_i^{-1}|) \\ &= \frac{\max(|\lambda_i|)}{\min(|\lambda_i|)} \end{aligned} \quad (4.19)$$

By definition

$$\max(\sigma_i) = \|A\| \quad (4.20)$$

and

$$\min(\sigma_i) = \max(\sigma_i^{-1}) = \|A^{-1}\| \quad (4.21)$$

Therefore,

$$\frac{\max(|\lambda_i|)}{\min(|\lambda_i|)} \leq \frac{\max(\sigma_i)}{\min(\sigma_i)} \quad (4.22)$$

■

4.4.2 Use of Non-optimal Bases

Once a badly conditioned linear system detected, we have to find an good over-approximation for the ill-conditioned optimal basis. An good approximation must satisfy: 1) the approximated cost is close to the optimal cost, 2) the corresponding basis is well conditioned 3) it is over approximated.

A square matrix is ill conditioned if at least one row (column) is nearly a linear combination of the other rows (columns). The number of rows (columns) that are nearly equal to linear combination of the remaining rows (columns) represents the *degree* of ill-condition. Note that changing any column in the ill-conditioned cycle changes the value of β_n and can remedy the problem. This situation is quite rare; so, we simply discard one of the columns from the standard form linear program and the linear program solver is restarted. This may result in a sub-optimal solution to the standard form LP and therefore a super-optimal solution to COHO-LP. This preserves our guarantee of overapproximation. To avoid excessive growth in the reachable region, we test the discarded constraint at the end of the LP solution. If it is clearly violated, then we bring that column back into the LP and reject a different one. We continue in this fashion until we find a good quality optimum.

The method has a geometric interpretation. For COHO linear programs, basic columns (rows) in the dual (primal) represent inward halfspace normals in the primal. The feasibility of a basis in the dual means that the primal cost vector is a positive combination of basis columns. Therefore, the primal cost vector lies inside the cone generated by the basic inward halfspace normals, also called the *basic cost cone*.

If the ill-conditioned basis is non-optimal, then our method will not affect

the final result. Let β be the set of feasible bases of the COHO dual linear program. Then once a badly conditioned basis found, some basic column c is excluded from the feasible basis. Therefore, the set of feasible bases β' for the modified linear program is the subset of β . Thus, our method only reduces the feasible region and the number of feasible bases. By the assumption that the ill-conditioned basis is not optimal, there exists a basic column c that is not in the optimal basis. Therefore, the optimal basis is not changed after excluding column c . Only an ill-conditioned optimal basis matters.

Now consider a two-dimensional COHO linear program. Here, there are only two basic columns, ill-conditioning indicates the two normals are nearly parallel, which means the optimal vertex is either highly obtuse or highly acute, as shown in Figure 4.3.

Because all bases visited in the Simplex algorithm are feasible for the dual², the optimization direction of the primal lies inside the optimal cost cone. Therefore, when the optimal vertex is very obtuse, i.e. the optimal cost cone is very acute, then the optimization direction is nearly perpendicular to sides of the optimal vertex. This suggests that vertices that lie on the lines corresponding to the two constraints are good choices for approximating the optimal cost. One constraint in the basis is dropped because it is redundant. The approximated vertex is obtained by replacing this constraint with some other constraint out of the basis in our algorithm.

For the highly acute vertex case, it is precluded in the geometry phase of COHO, which replaces such polygons with a small rectangle. Thus it is not considered.

Now, let us try to extend the idea to higher dimensions. Consider a COHO linear program of dimension d whose optimal basis \mathcal{B} is ill-conditioned. Let us assume the degree of ill-condition is one, it is straightforward to generalize to higher

²For our solver, it is possible that little infeasible is visited. However, if the infeasible basis is ill-conditioned, it is safe to drop it because the basis is infeasible which should not be visited at all.

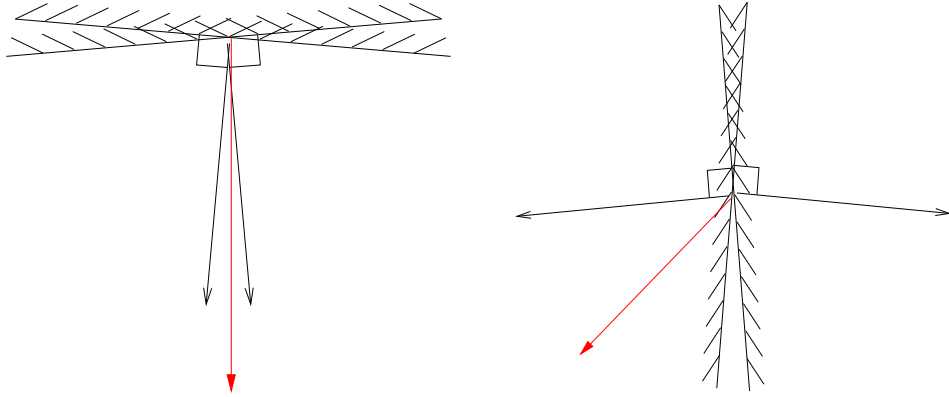


Figure 4.3: Types of Optimal 2D Vertices: 1)highly obtuse optimal vertex and highly acute cost cone; 2)highly acute optimal vertex and highly obtuse cost cone.

degrees.

Let G_k be the linear subspace generated by all basic columns except the k th: $\{\mathcal{B}_1, \dots, \mathcal{B}_{k-1}, \mathcal{B}_{k+1}, \dots, \mathcal{B}_d\}$. The subspace is $d - 1$ dimensional, i.e., a hyperplane. Each of these hyperplanes generated in the manner is a face of the optimal cost cone. Assume that the last basic vector \mathcal{B}_d is nearly a linear combination of the other columns, while the others are independent. Therefore, the projection of \mathcal{B}_d onto direction n_d , which is the norm of G_d and also the intersection of faces corresponding to the $1^{st}, \dots, d - 1^{th}$ constraints, is almost zero.

As shown in Figure 4.4, if the basis vector \mathcal{B}_d onto the hyperplane G_d is positive, then the cost cone is highly acute and nearly $d - 1$ dimensional. Therefore, the optimization direction v is almost on the $d - 1$ dimensional hyperplane; it is almost perpendicular to vector n_d . Therefore, any vertex on the norm of G_d is a good approximation. The d^{th} constraint is dropped to find the approximated vertex similar with the two dimensional case.

Otherwise, if the projection of basis vector \mathcal{B}_d onto the hyperplane G_d is negative, the cost cone is highly obtuse and it is also nearly $d - 1$ dimensional. The angle between \mathcal{B}_d and the norm of G_d is highly acute. However, in practice such

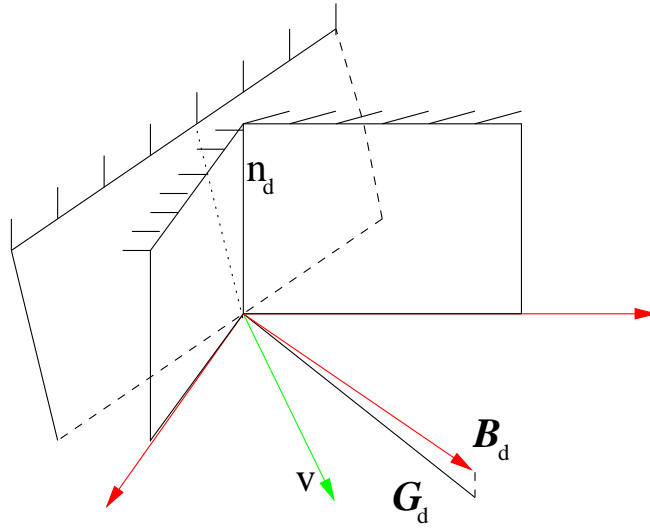


Figure 4.4: Highly Obtuse Optimal 3D Vertices

cases are removed similarly in the geometry step.

Therefore, our method is reasonable and practical. The error introduced by the approximation is quite small in our applications.

4.5 Implementation

The COHO solver is implemented in `coho.lp` and `coho.lp.solver` packages. The `lp` package defines the interface for a linear program, including classes for linear program, constraints, basis and result. It also defines a interface for linear program solver, each solver should implement it. This architecture makes it quite easy to change to a new linear program solver. The `solver` package implement the linear system solver in the `CohoMatrix` class and the linear program solver in the `CohoSolver` class. The solver is integrated with interval number implemented in `coho.common.number` package describe in Chapter 3.2. The `CohoMatrix` is a subclass of `DoubleIntervalMatrix` in `coho.common.matrix` package. The structure of COHO solver is shown in Figure 4.5.

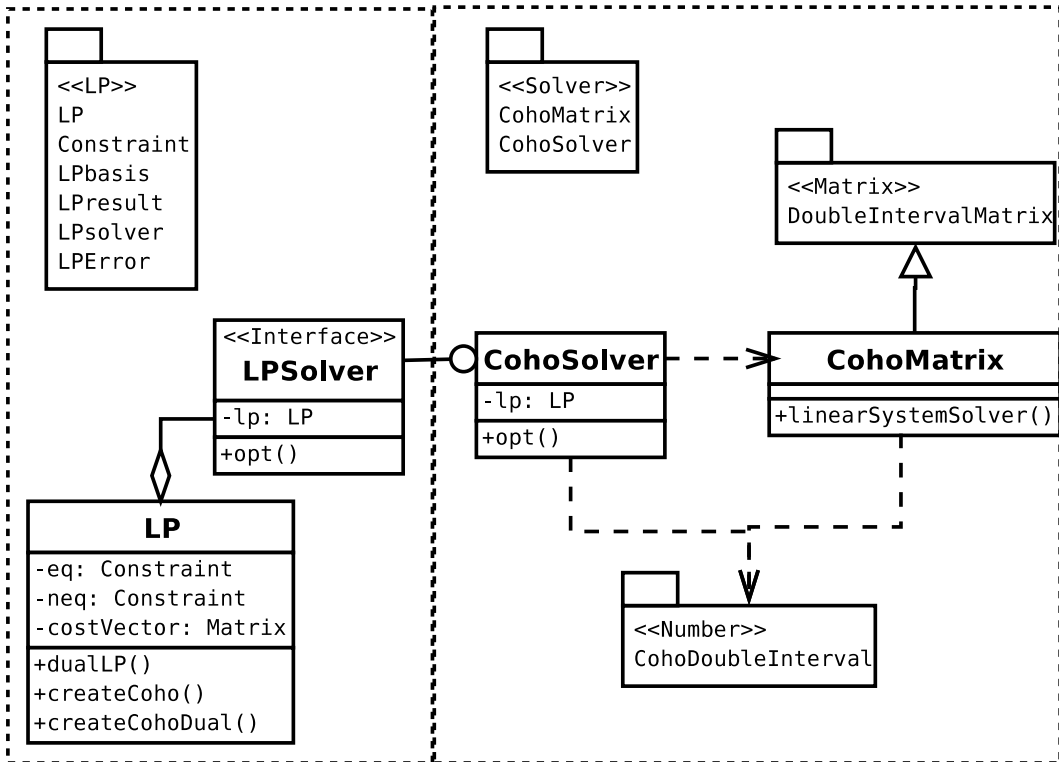


Figure 4.5: Class Hierarchy of COHO Linear Program Solver

In section 4.2.3, the *Big M method* is described. This algorithm guarantees finding an initial feasible basis for any linear programs, however, it does not use the special structure of COHO linear programs. In the implementation, a fast method is applied first, the Big M method is used only if it fails. The idea is very simple: For the standard COHO linear program, there are at most two nonzero elements in each column, and usually there are many columns that have only one nonzero element. Therefore, the solver tries to find a basis that there is only one nonzero element in each row and each column. It's obviously that this kind of bases is invertible and feasible. If such a feasible basis is found, no extra variables need to be introduced, which makes the following pivoting much easier and faster.

If *Big M method* is performed, extra variables are added to the linear pro-

gram. There might be several ill-conditioned bases encountered during the pivoting to drive out these extra variables. Because the weight for extra variables are expensive, the initial feasible basis may lead to some bases that have very large cost while still clearly favorable for pivoting, for example, two nearly parallel edges for the 2D case. However, this will introduce some unexpected ill-condition bases, which slows our algorithm. To avoid such kind of badly conditioned basis, extra variables are never allowed to enter into the basis once driven out in the pivoting. And if a new favorable basis is obtained by driving out one extra variable, the conditioning of the new basis is checked. If it is highly ill-conditioned, the algorithm tries to find a new well-conditioned while clearly favorable basis as possible. This removes lots of redundant badly conditioned bases.

The COHO solver is complex to debug, it depends on several packages, including the `number` and `matrix` packages. However, it is easy to check the optimization of a basis. The solver and related packages are validated by checking the feasibility of bases returned by solver for both primal and dual linear programs.

Chapter 5

Projection

At each step of Coho, the advanced projectahedron must be projected onto the 2D subspace, as described in Chapter 2. Each time advanced slab is described as a series of linear inequalities, i.e. the feasible region of a COHO linear program. Producing the projection polygon from these linear inequalities requires solving a series of linear programs using our COHO solver.

Coho's algorithm for computing projection polygons is introduced in section 5.1. Section 5.2 describes the issues we encountered for a practical implementation and reducing error. Our algorithm for converting a general projection problem to the simple one is described in section 5.3. Section 5.4 analyzes the error of the COHO projection algorithm.

5.1 Algorithm

At the end of a time step, a slab is advanced to a projectahedron according to the linear model. To maintain the projectahedron representation, the projection onto the original 2D subspace is needed. Formally, the problem is to project a projectahedron described as:

$$P \cdot x \geq b \tag{5.1}$$

where P is a COHO matrix, onto projection plane \mathcal{P} , generated by basis vectors $e_i, e_j \in \{\|e_i\| = \|e_j\| = 1, e_i \cdot e_j = 0\}$. It is a special case of computing the projection polygon of a polyhedron onto a 2D subspace.

The basic idea is to find points on the projection polygon by maximizing the value along each direction on the projection plane. The optimization problem is to solve a linear program, of which the feasible region is the projectahedron and the optimization direction is the vector on the projection plane, for example, the normal of an edge. It can be solved by the COHO solver. However, there are an infinite number of possible directions on the projection plane. It is impossible and unnecessary to solve the optimization problems for every possible direction. To reconstruct a polygon, only the polygon vertices are required. As shown in Figure 5.1, solving the optimization problems along the normals of polygon edges is enough to find all polygon vertices and the projection polygon.

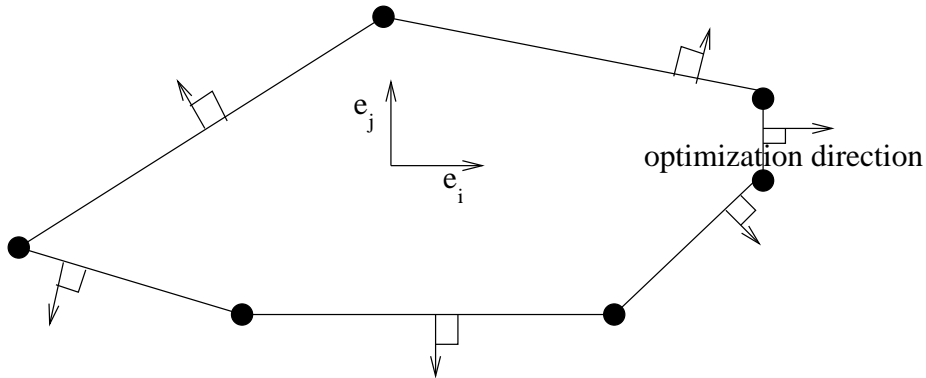


Figure 5.1: Projection Algorithm Using Linear Programming: polygon vertices are found by optimizing along the normal of edge.

Our algorithm finds the normal for each edge of the projection polygon in a counter clockwise sweep. Given the right endpoint of the previous edge, i.e, the left endpoint of current edge, the normal of the current edge is computed using the feasibility of optimal basis returned from previous linear program; then a linear program, of which the optimization direction is the normal, is constructed, the right

endpoint of current edge is found by solving it. This step, as shown in Figure 5.2, is performed until all edges are found, then the projection polygon is reconstructed easily.

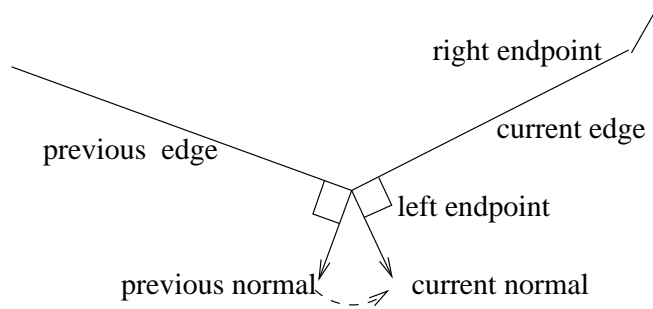


Figure 5.2: Find the Normal of Current Edge Using Optimal Basis of Previous Linear Program

The essential part of our algorithm is the *currNorm* function which computes the normal of the current edge. Given the linear program, of which optimization direction is the normal of the previous edge, the optimal basis \mathcal{B} that corresponds to the left endpoint of the current edge is optimal for any linear programs with optimization direction between the *previous normal* and the *current normal*. As shown in Figure 5.2, when the optimization direction exceeds the *current normal*, \mathcal{B} is no longer optimal. It becomes a feasible but non-optimal basis for the COHO linear program and thus infeasible for the COHO dual linear program. Therefore, the *current normal* is the critical direction that forces \mathcal{B} to be infeasible for the standard form COHO linear program.

For the standard form COHO LP, the infeasible basis forces at least one variable to become negative. Let $e_i + \alpha e_j$ be the optimization direction, for a basis \mathcal{B} , the variables x are given by:

$$\begin{aligned}
 x &= P_{\mathcal{B}}^{-T} \cdot (e_i + \alpha e_j) \\
 &= P_{\mathcal{B}}^{-T} e_i + \alpha P_{\mathcal{B}}^{-T} e_j \\
 &= \pi + \alpha \eta
 \end{aligned} \tag{5.2}$$

where $\pi = P_{\mathcal{B}}^{-T} e_i$ and $\eta = \alpha P_{\mathcal{B}}^{-T} e_j$. Obviously, the α value of *current normal* is the smallest that makes some variable x_i of x nonpositive:

$$\alpha_{currNorm} = \min_i \left(-\frac{\pi_i}{\eta_i} \right) \quad (5.3)$$

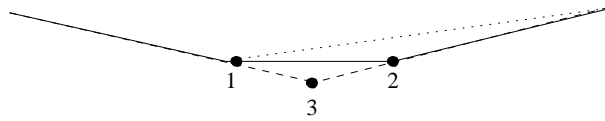
Here, π_i should be positive and η_i should be negative. Once the direction exceeds *current normal*, the α value is greater than $\alpha_{currNorm}$, then the variable x_i is negative, and \mathcal{B} is no longer feasible.

Of course, any optimization direction which is ahead of the *previous normal* also makes the basis \mathcal{B} infeasible. This can be easily avoided by forcing the α value to be greater than the $\alpha_{currNorm}$.

Given an optimization direction, the corresponding linear program is solved; both the optimal basis and optimal value are returned by the COHO solver. A polygon can be represented as a collection of edges or a set of vertices. Here, the first method is used and an edge is represented by the optimal direction and its optimal value. The vertices are reconstructed from edges at the final step; rather computing from optimal bases directly, although they are usually the same. There are several reasons for choosing this approach. First, the optimal basis returned from the COHO solver is *possibly* optimal, it might not be the exact vertex, even though it will be quite close. Second, if some edge is omitted, our method guarantees the final result is over approximated, while it is under approximated using the vertices directly. As shown in Figure 5.3, if the middle edge is skipped, our method will replace the 1st and 2nd vertices with the 3rd vertex. Otherwise, the 2nd vertex is ignored and an unacceptable under approximation is produced.

The basis corresponding to right endpoint of current edge should be found to compute the normal of the next edge, as shown in Figure 5.2. Ideally, if the optimization direction is over the normal, the optimal basis for the linear program is the one corresponding to the right endpoint. However, for our COHO solver, only *possibly* optimal bases are available. Therefore, to drive out the left endpoint from the set of *possibly* optimal bases, the optimization direction should be slightly

under approximation when the vertex of the middle edge is skipped



over approximation when the normal of the middle edge is skipped

Figure 5.3: Overestimation of Projection Polygon When an Edge is Skipped

greater than the *current normal*. Of course, this may skip some edges, but it only introduces small amount of over approximation, which will be proved in section 5.4.

5.2 Implementation

The projection plane is partitioned into four quadrants by basis vectors e_i and e_j . The normal of polygon edge is represented as combination of basis vectors. As shown in Figure 5.4, it's $e_i + \alpha e_j$ in the first quadrant, $e_j - \alpha e_j$ in the second quadrant and so on, and α is always positive. The reason for this representation is that it's easy to compute the normal of the edge, as shown in equation 5.3.

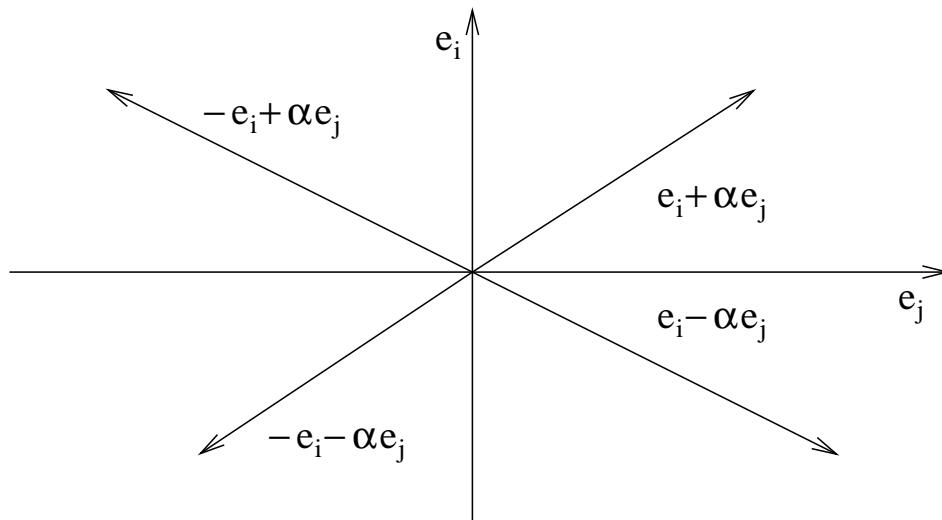


Figure 5.4: The Representation of Optimization Direction

Because of round off error and error from the COHO linear program solver, the optimization direction might not be the exact normal of the polygon edge. If the adjacent normals are nearly the same, i.e. the adjacent polygon edges are nearly on the same line; it's better to replace the two normals with one and combine the two edges as one. This will introduce small error, which will be examined in section 5.4.

In the implementation, we define a threshold ϵ , the *currNorm* function will force the angle of normals of current and previous edges to be bigger than a small amount ϵ . If the angle is less than ϵ , these two normals are combined and one is dropped. When the normal is close to the second basis vector, i.e. α is close to infinity, it is replaced with the second basis vector, which prevents large computation error caused by a huge value for α .

As initialization of our algorithm, the first optimization direction is required. A linear program with optimization direction e_i is solved first and the basis for the left endpoint of the first edge is found. Using this basis, the normal for the first edge is found, as shown in Figure 5.5.

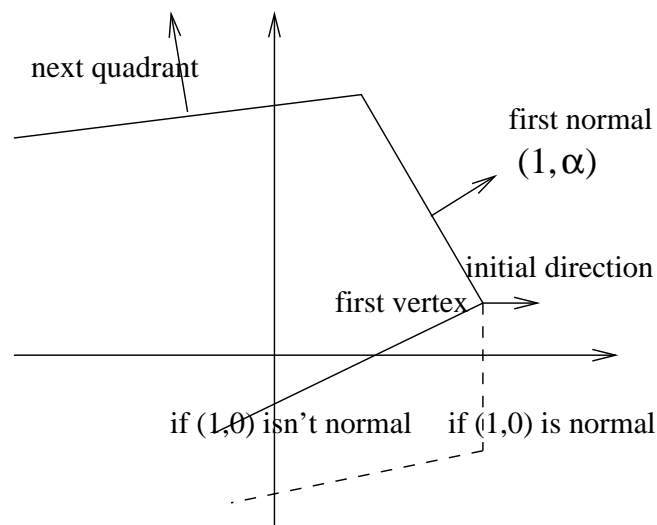


Figure 5.5: Find the Initial Optimization Direction for Projection Algorithm

However, the initial optimization direction might be the exact normal of the

first edge, as shown in Figure 5.5. In this case, the first edge is omitted. Therefore, at the end algorithm enters into the first quadrant again and check whether it's an exact normal or not. If it is, the first edge is added.

To find the right endpoint, the normal of current edge is moved forward by a small amount ϵ , then a linear program is solved to maximize value along this direction. This may omit some edges, but the error is small. However, the left endpoint may also be in the set of possible optimal bases. We will turn the direction increasingly until the left endpoint is driven out or the change in angle exceeds a maximum angle.

If the normal of the current edge is in the next quadrant, increasing α will not change the optimal basis. Thus the *currNorm* can not find a value for α that makes the optimal basis infeasible for the standard COHO linear program, the program should jump to the next quadrant and reset the value α . The initial α should be zero; however, in the implementation, it is assigned with a negative value. The *currNorm* function forces α to be non-negative.

However, the *nextNorm* function might return 'jump to next quadrant' incorrectly. This is caused by the COHO solver. As described above, the right endpoint of the current edge is required to compute the normal of the next edge. However, the left endpoint might be in the set of possible optimal bases. If the left endpoint is used, the α value for current normal is returned again, it is not bigger than the current α by the required amount, ϵ , therefore, the algorithm jumps to the next quadrant.

This special case will introduce large amount of over approximation error. To reduce it, backtracking is employed when jumping to the next quadrant. As shown in Figure 5.6, the angle between current normal and the second basis vector is bisected, the new direction is used as the optimization direction and the optimal vertex is found. If the *currNorm* performed correctly and there is no edge omitted, the result should be the same with that of the previous linear program, then algorithm jumps

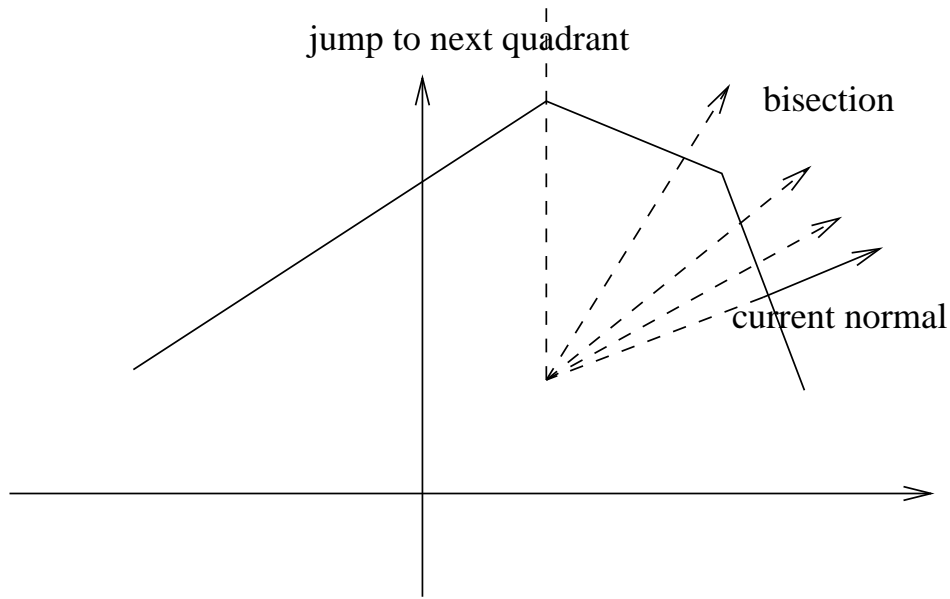


Figure 5.6: Backtracking to Reduce Error When Jump to Next Quadrant

to the next quadrant. Otherwise, some edges were skipped and a new optimal basis that is far from the current optimal basis should be found. The algorithm tries to find the edges omitted: it bisects the angle between current normal and the second basis vector and moves the optimization direction back toward the current normal until the linear program has the same optimal basis with a current basis or the angle is not bigger than angle of current normal by a small amount ϵ .

There is also a special case when the algorithm jumps two quadrants in a single step, for example, the normal of current edge is in the first quadrant while the normal of next edge is in the third quadrant. In such case, *currNorm* might perform incorrectly. The solution is to preclude this special case. As shown in Figure 5.7, an extra edge in the middle quadrant $ax + by = c$ is added. The normal of the edge is $y - x$ which is obviously in the middle quadrant and the value of c is $v_{opt} - v_{const}$ where v_{const} is a proper positive constant and v_{opt} is the optimal value for the linear programs with optimal direction as $y - x$. As the result, there are

no two quadrants jumps and the feasible region is reduced. However, this is not an under approximation because at the reconstruction step as describe in the following paragraph, this edge is removed.

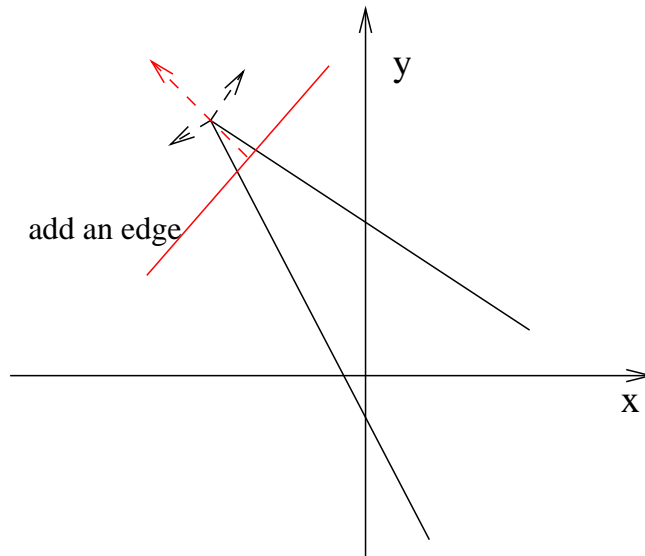


Figure 5.7: Special Case of Jumping Two Quadrants Continuously

At last, a collection of edges is found. To get the vertices of the polygon, the intersection point of adjacent edges are computed. They are solved by our linear system solver efficiently because they are COHO linear systems.

5.3 Converting from End-of-Step to Beginning-of-Step Projection

The previous section described projection for projectahedra described by equation 5.1. After moving forward according to the linear model

$$x' = Mx + q \tag{5.4}$$

the forward projectahedron is describe as

$$P \cdot E^{-1}x \geq b \quad (5.5)$$

where $E = e^{M\Delta t}$ is the forward Euler operator. In general, PE^{-1} is no longer a COHO matrix which prevent us from using our fast linear system solver to compute π, η and reconstruct the vertices at the last step of projection.

The solution is converting the problem of projecting $P \cdot E^{-1}x \geq b$ onto the projection plane generated by basis vectors e_i and e_j to the problem of projecting $Px \geq b$ onto projection plane generated by basis vectors Ee_i and Ee_j . The polygon vertex for the original problem p and its corresponding polygon vertex for the converted problem p' have the relation

$$p = Ep' \quad (5.6)$$

The method has an algebraic explanation proposed by Marius [Laz01]. In the projection step, given an optimization direction d , a linear problem

$$\begin{aligned} & \min(d^T \cdot x) \\ \text{s.t. } & P \cdot E^{-1}x \geq b \end{aligned}$$

should be solved. Let $y = E^{-1} \cdot x$ we have

$$\begin{aligned} & \min((E^T \cdot d)^T \cdot y) \\ \text{s.t. } & P \cdot y \geq b \end{aligned}$$

Therefore, linear programs for original and converted projection problems have the same optimal value and $y_{opt} = E^{-1}x$. Our method is valid.

The method also has a geometric explanation. Let us assume that a polyhedron is represented as $P \cdot E^{-1}x \geq b$ in the coordinate \mathcal{C} with basis $(e_1, \dots, e_n)^T$, and the projection plane is generated by e_i and e_j . Now, in the new coordinate \mathcal{C}' with new basis defined as $E(e_1, \dots, e_n)^T$, the representation for the same projectahedron

should be $Py \geq b$, and the basis vectors for the projection plane should be Ee_i^T and Ee_j^T . In the new coordinate \mathcal{C}' , the vertex of projection polygon is computed as p' , its representation in coordinate \mathcal{C} should be $p = Ep'$. Therefore the two problems must have the same projection polygon because they are simply different representations of the same polyhedron and projection plane.

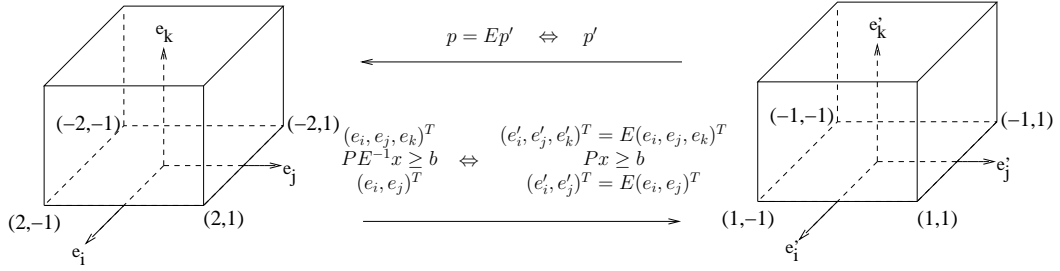


Figure 5.8: Conversion between End-of-Step Projection and Beginning-of-Step Projection

Figure 5.8 illustrates this conversion. In the example, the basis for the coordinate system \mathcal{C} is

$$(e_1, e_2, e_3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.7)$$

and

$$P = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{pmatrix}, E = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, b = \begin{pmatrix} -1 \\ -1 \\ -1 \\ -1 \end{pmatrix} \quad (5.8)$$

The projection plane is the one generated by e_1, e_2 . Then the new coordinate \mathcal{C}' should be:

$$(e'_1, e'_2, e'_3) = E(e_1, e_2, e_3)^T = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.9)$$

The polyhedron is of the simple form $P \geq b$. The basis for projection plane in \mathcal{C}' is changed to $e'_1 = (2, 0, 0)^T$ and $e'_2 = (0, 1, 0)^T$. Now, in \mathcal{C}' it is easy to see the projection polygon with vertices

$$p' = \begin{pmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (5.10)$$

Now multiply it with E to obtain

$$p = Ep' = \begin{pmatrix} 2 & -2 & -2 & 2 \\ 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (5.11)$$

which are exactly the vertices of projection polygon in the original coordinate \mathcal{C} .

5.4 Error Analysis

Our projection algorithm overestimates the projection polygon, however, the error introduced is quite small.

First, the COHO linear system solver returns possibly optimal vertices. Such a vertex might be infeasible for the COHO linear program. Assume the maximum distance from the possibly optimal vertex to the feasible region of COHO linear program is γ . In other words, the COHO solver can clearly detect the infeasibility of points for which the distance to the feasible region is greater than γ .

As shown in Figure 5.9, the projection algorithm finds a vertex first, then the corresponding edge is computed; with the normal of the edge, a new linear program is constructed and its optimal vertex is solved as the vertex in the next step, and so on. The final polygon computed might be different from the correct one. It might have some edges that are close to the correct instead, or have some extra edges. However, if no edge is skipped, all polygon vertices of the overestimated polygon are optimal vertices of corresponding linear programs, and the distance to the feasible

region is less than γ . It is obvious that the maximum increase in area caused by the COHO solver is γL where L is the perimeter of the projection polygon.

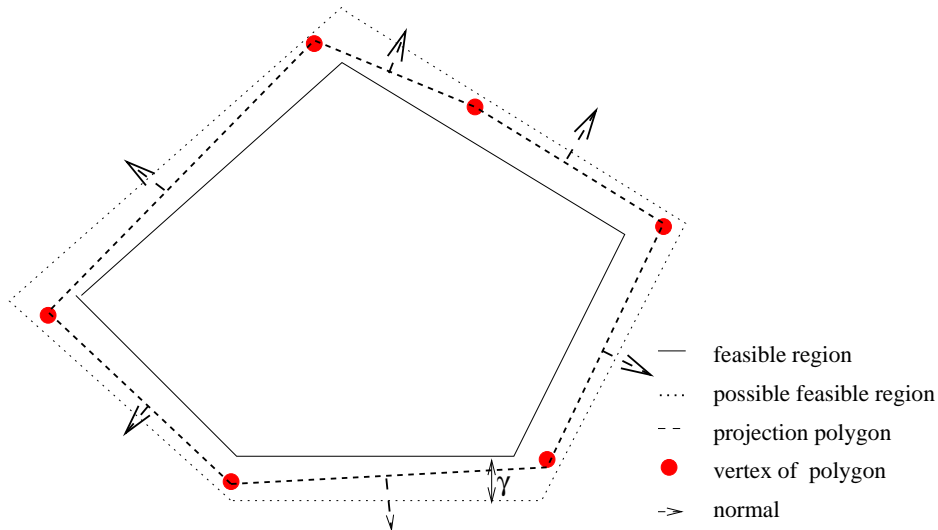


Figure 5.9: Projection Error Introduced by COHO Linear Program Solver

However, it is possible that some edges are skipped. In this case, the polygon vertex might be new point created by intersecting two edges, rather than optimal vertex of COHO linear program. Such vertices are clearly infeasible; in which cases the error is not bounded by γL as described above. There are three cases in which edges might be skipped:

1. As desired, our algorithm skips the edge if the angle between it and the previous edge is less than ϵ , as shown in Figure 5.10. It is easy to compute that the maximum possible error introduced is $\frac{1}{2}l^2 \sin \epsilon \approx \frac{1}{2}l^2 \epsilon$, where l is the length of edge skipped. Therefore, the maximum total error of this kind is less than $\frac{1}{2}L^2 \epsilon$.
2. Notice that our algorithm uses the left endpoint to find the normal of the edge. However, the COHO solver might return points close to the left endpoint as the optimal vertex, and the normal computed might differ from the correct one. If

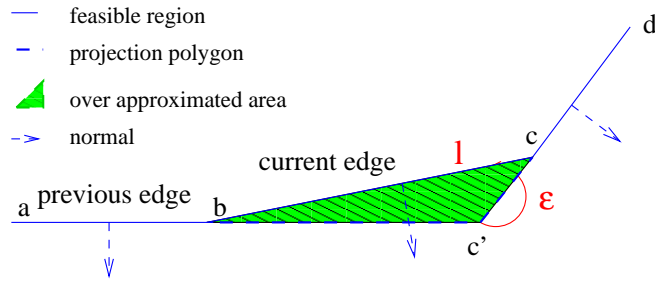


Figure 5.10: Projection Error Introduced by Ignoring Small Angle

it is smaller, no edge is skipped, but some redundant edges may be introduced. If it is greater, some edges might be skipped. However, the computed normal is greater than the correct normal by no more than $\beta = \arctan \frac{\gamma}{l}$ as shown in Figure 5.11. Usually this angle is smaller than ϵ , and the error is included in $\frac{1}{2}L^2 \sin \epsilon$ described above. If the angle is greater than ϵ , then the edge is

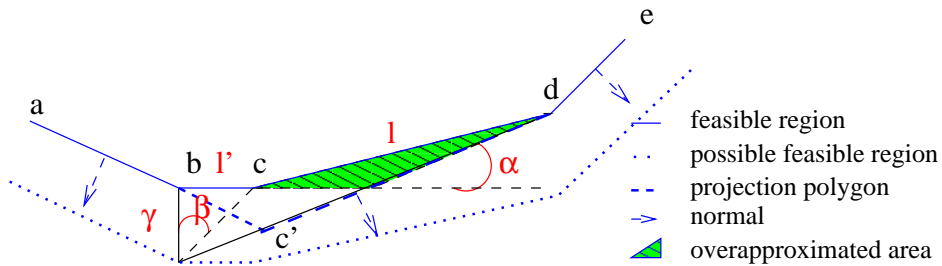


Figure 5.11: Projection Error Introduced by Skipping Short Edge

very short because from $\frac{\gamma}{l} \approx \beta = \arctan \frac{\gamma}{l} \geq \epsilon$, we know l' is bounded by $\frac{\gamma}{\epsilon}$ which is a quite small value in practice. It can be computed that the total error of this kind is bounded by $\frac{1}{2}\gamma L$. In fact, the error is counted in the error γL caused by the COHO solver; because no new points are introduced in the computed polygon in this case, although some edges are skipped. All vertices are from the COHO solver's optimal result. Therefore no vertex is far from the feasible region by γ , each vertex is in the possible feasible region of COHO solver.

3. Although it occurs rarely, the COHO solver might return the left endpoint (or nearby) of the previous edge as the optimal vertex. In this case, the normal computed might be less than the previous normal, hence algorithm tries to jump to the next quadrant. As described earlier, backtracking is employed to reduce the error. It stops only if there is new vertex found if the angle is greater than the previous normal by 2α , while there is no new vertex if it is greater by only α . In this case, the maximum possible area increase is less than $\frac{1}{2}l^2 \sin 2\alpha$, where l is the distance between the new vertex and previous vertex, as shown in Figure 5.12. Furthermore, l is less than a small constant c , otherwise, bisection is performed between α and 2α to find the edges omitted. Therefore, the area increase by jumping to next quadrant is bounded by a small constant. In fact, if efficiency is not considered, bisection can be performed until no edge is skipped, then no error is introduced.

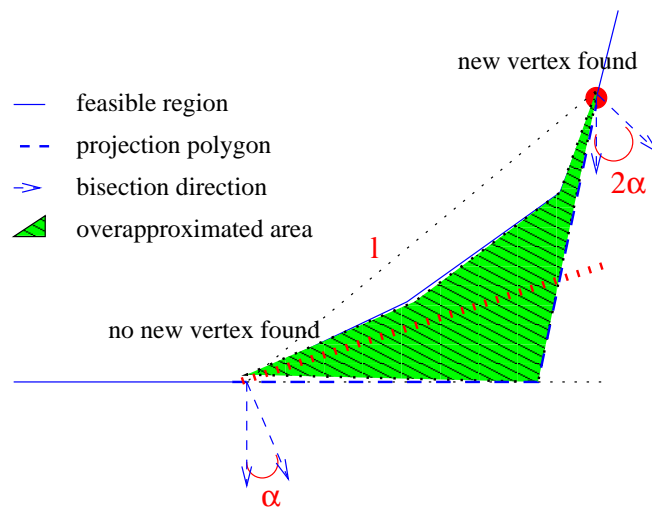


Figure 5.12: Projection Error Introduced by Jumping to Next Quadrant

Therefore, the maximum possible error for our projection method is about $\gamma L + \frac{1}{2}L^2 \sin \epsilon$. ϵ is a constant in the algorithm and it is set to $1e - 12$ in the current implementation. We believe γ is quite small based on the current experimental

result reported in Chapter 6. Thus the error is quite small in practice.

Chapter 6

Experiments

This chapter shows two examples of COHO: a two dimensional Sink example and a three dimensional Van der Pol oscillator; these demonstrate the correctness of COHO and robustness of the implementation. The performance of the linear program solver and projection algorithm is analyzed from the experimental data.

6.1 Application of Coho Verification System

We first applied COHO to the *Sink* example from [DM98], a two-dimensional, linear system. The example has the same dynamics model with Dang and Maler's:

$$\begin{aligned} \dot{x} &= -2x - 3y \\ \dot{y} &= 3x - 2y \end{aligned} \tag{6.1}$$

However, to prove that COHO has the capability to deal with more complex system, we have changed the initial region from the rectangle with the diagonal $[(0.1, 0.1), (0.3, 0.3)]$ to a polygon with vertices

$$[(0.1, 0.1), (0.3, 0.1), (0.4, 0.4), (0.1, 0.3)]. \tag{6.2}$$

The reachable space is shown in Figure 6.1.

The result is much more accurate compared with the result from Figure 5.4) in [DM98]. Our result clearly shows the boundary and distinct cycles of the

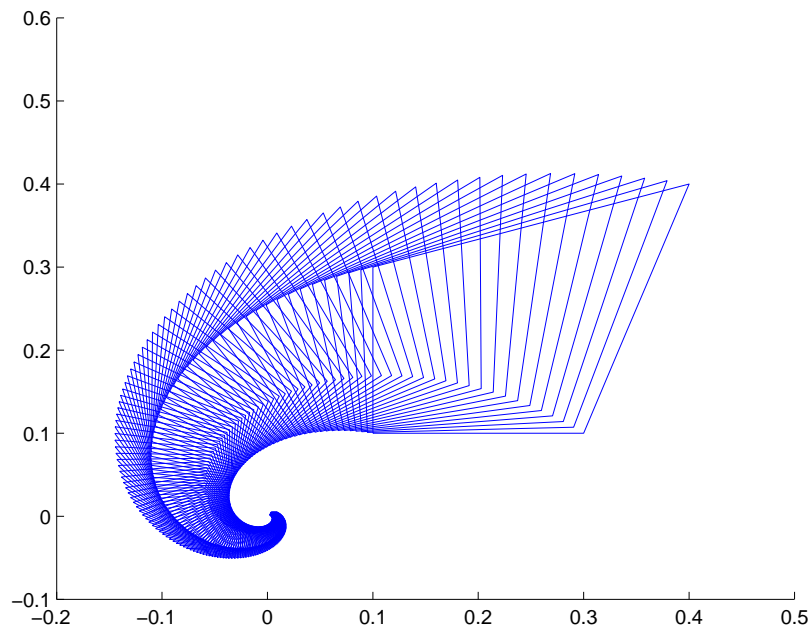


Figure 6.1: The Result of a Two Dimensional Linear Model Example: Sink

spiral, whereas Dang and Maler’s result merges them together and has much more overapproximated space. The result is even better than that of the earlier version COHO, as shown in Figure 4 of [GM99]. In the earlier version, the spiral converges to a small circular shape; whereas in the current version, the reachable set of the last step reaches a much smaller polygon that is nearly a point as seen in the Figure 6.1. This demonstrates that the over approximation is much smaller than before. The greater accuracy of new COHO arises mainly from two factors. First, the COHO linear program solver offers tighter over approximations of the optimal value, which is mainly because of its handling of badly conditioned problems. Second, the projection algorithm is more accurate; it skips fewer edges than the earlier version and thereby produces more accurate projection polygons. Although the Sink example is two-dimensional and thus no projection is taking place; the projection

algorithm is performed to compute the polygon representation of the reachable space from the linear program which represents the advanced face at step 4(a)iv described in Section 2.4.

The reachable space of the Sink example above is two dimensional and the system dynamics are linear. Therefore, it does not use the projectagon technique. We now apply COHO to a three-dimensional, non-linear model to demonstrate the reachability calculation using projectagon techniques. The model is derived from a Van der Pol oscillator (3VdP) [HS74, pp. 217f] where we have added the z variable to provide an example with three dimensions and demonstrated COHO's projection capability.

The model is described by the following nonlinear ODE:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -y - x^3 + x \\ x - y^3 + y \\ 2x^2 - 2z \end{pmatrix} \quad (6.3)$$

and the initial region is the hypercube with the diagonal

$$[(1.0, -0.05, 0.9), (1.2, 0.05, 1.1)] \quad (6.4)$$

Because the model is nonlinear, we start by approximating it with an linear differential inclusion. By Taylor's theorem, $\forall x \otimes y \otimes z \in [x_{lo}, x_{hi}] \otimes [y_{lo}, y_{hi}] \otimes [z_{lo}, z_{hi}]$, the function can be approximated as:

$$f \begin{pmatrix} x \\ y \\ z \end{pmatrix} \approx f \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{pmatrix} + J \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{pmatrix} \cdot \begin{pmatrix} x - \bar{x} \\ y - \bar{y} \\ z - \bar{z} \end{pmatrix} \quad (6.5)$$

where J is the *Jacobianian Matrix* and $\bar{x} = (x_{lo} + x_{hi})/2$, $\bar{y} = (y_{lo} + y_{hi})/2$, $\bar{z} = (z_{lo} + z_{hi})/2$. After computing the Jacobian Matrix, the equivalent converts to:

$$f \begin{pmatrix} x \\ y \\ z \end{pmatrix} \approx \tilde{f} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 - 3\bar{x}^2 & -1 & 0 \\ 1 & 1 - 3\bar{y}^2 & 0 \\ 4\bar{x} & 0 & -2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} 2\bar{x}^2 \\ 2\bar{y}^2 \\ -2\bar{z}^2 \end{pmatrix} \quad (6.6)$$

Now, let us find the error bound:

$$\Delta f = |f - \tilde{f}| = \begin{pmatrix} |x^3 - 3\bar{x}^2x + 2\bar{x}^3| \\ |y^3 - 3\bar{y}^2y + 2\bar{y}^3| \\ |2(x - \bar{x})^2| \end{pmatrix} \quad (6.7)$$

Let $\Delta x = x - \bar{x}$, and we rewrite $x^3 - 3\bar{x}^2x + 2\bar{x}^3$ as $3\bar{x}(\Delta x)^2 + (\Delta x)^3$. Noting that $\Delta x \in [-\frac{x_{hi}-x_{lo}}{2}, \frac{x_{hi}-x_{lo}}{2}]$, it follows that the maximum value of $|x^3 - 3\bar{x}^2x + 2\bar{x}^3|$ occurs for $\Delta x = \frac{x_{hi}-x_{lo}}{2}\text{sign}(\bar{x})$. At this point $|x^3 - 3\bar{x}^2x + 2\bar{x}^3| = \Delta x(\Delta x + 3|\bar{x}|)$. Similar reasoning applies to the other two terms of Δf , and we conclude:

$$\max(\Delta f) \leq \begin{pmatrix} \Delta x^2(\Delta x + 3|\bar{x}|) \\ \Delta y^2(\Delta y + 3|\bar{y}|) \\ 2\Delta x^2 \end{pmatrix} \quad (6.8)$$

where $\Delta x = (x_{hi} - x_{lo})/2$, $\Delta y = (y_{hi} - y_{lo})/2$, $\Delta z = (z_{hi} - z_{lo})/2$. This yields the differential inclusion:

$$f \begin{pmatrix} x \\ y \\ z \end{pmatrix} \subseteq \begin{pmatrix} 1 - 3\bar{x}^2 & -1 & 0 \\ 1 & 1 - 3\bar{y}^2 & 0 \\ 4\bar{x} & 0 & -2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} 2\bar{x}^2 \\ 2\bar{y}^2 \\ -2\bar{z}^2 \end{pmatrix} + \text{cube} \begin{pmatrix} \Delta x^2(\Delta x + 3|\bar{x}|) \\ \Delta y^2(\Delta y + 3|\bar{y}|) \\ 2\Delta x^2 \end{pmatrix} \quad (6.9)$$

COHO uses this approximated dynamic model to compute the reachable state space. Figures 6.2 and 6.3 show the projections onto the x - y and x - z planes. Note that in both projections, the region at the end is contained in the initial region. This establishes the invariance of the computed region. All trajectories from this invariant set remain in the set for all time.

The example is new; thus it is hard to find other results with which to compare our result. We try to compare our result with the two dimensional Van der Pol oscillator example in [GM99]. The models for variables x and y are the same, therefore we can compare the projection of our reachable space onto the x - y plane with the result from Greenstreet99a. From Figure 6.2, it is easy to see

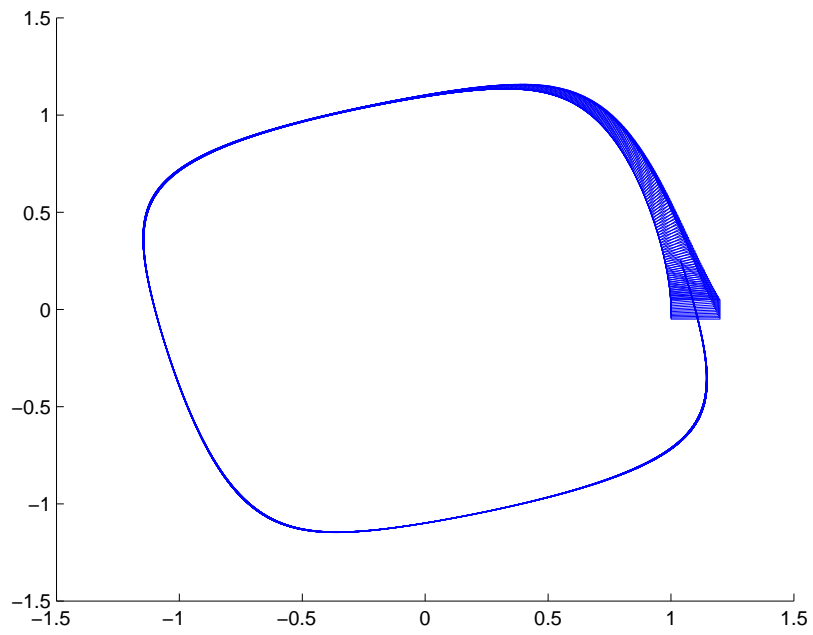


Figure 6.2: The Result of 3VdP Example: $x-y$ plane

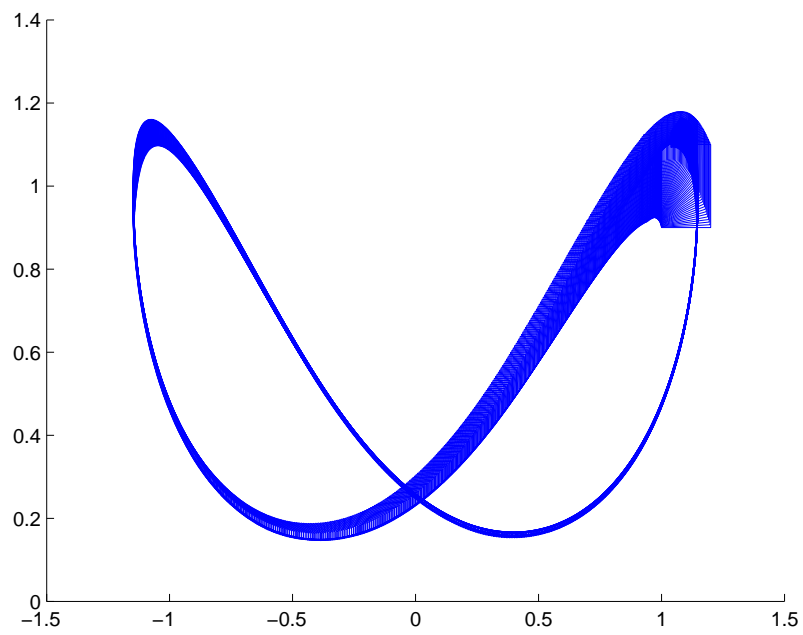


Figure 6.3: The Result of 3VdP Example: $y-z$ plane

that the trace converges much faster. In [GM99], the reachable set is computed only for the upper half cycle; the lower half cycle is the same with the above half by symmetry. However, the reachable set for the whole cycle is computed in our example. Although it is not necessary to verify the safety properties, it demonstrates that our algorithm and implementation are much more robust than the original version of COHO. The use of symmetry in the original version was because the over-approximation of the reachable space became excessive as the projectagon went around the subsequent corners. Furthermore, the original version was not able to analyse the three dimensional oscillator ([GM99] presents the more standard two-dimensional version) because of numerical stability issues. We have solved all of these problems.

6.2 Experimental Result for LP solver and LP project

The Sink and 3D Van der Pol oscillator examples require a large number of linear program solutions and polygon projections. For example, there are 87148 linear programs solved totally in the Sink example and 2051893 linear programs solved in the 3D Van der Pol oscillator example with more than 1000 time advance steps. The success of verifying these examples demonstrates the soundness of our linear program solver and projection algorithms, the efficiency of the projectagon representation of reachable states and the tightness of our over approximation from the linear program solver, the projection algorithm and modeling linearization.

First, we examine the error (the width of the interval for the cost) of the optimal value produced by the linear program solver. As seen in Figure 6.4, most of results have relative error less than 10^{-14} , and almost all optimal results have error bounds less than 10^{-7} . We observe that the relative error is approximately bounded by the \sqrt{ulp} (*unit of least precision*), A double precision ulp is 2^{-53} which is roughly 10^{-16} .) Figure 6.4 shows the distribution of relative error of the optimal results for both examples.

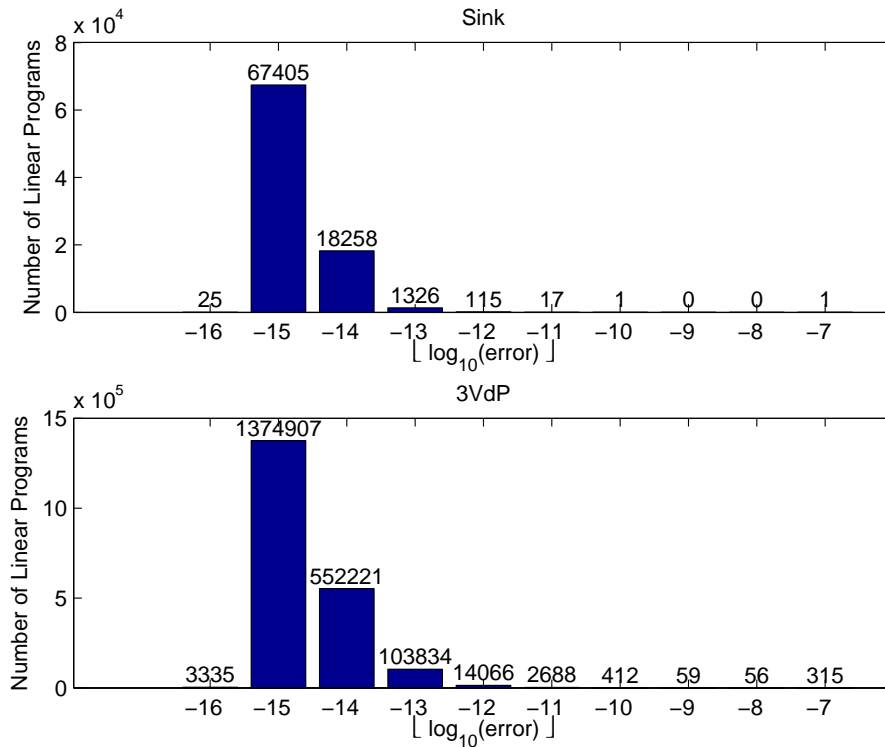


Figure 6.4: Error Distribution of Optimal Result of COHO Linear Program Solver

Next, we examine the pivot selection method that we presented in Chapter 4.2.1. In the Sink example, each linear program is solved with about 4.67 pivots, while the number is much larger in the 3VdP example, it is about 8.3 pivots per linear program. This is because the 3VdP example is three dimensional and has more constraints for each linear program; therefore, there are many more feasible vertices to visit before reaching the optimal point. We observe that most pivots have a unique more favorable basis, which demonstrates the efficiency of our pivot selection rules. Figure 6.5 shows the distribution of the number of branches per pivot. In the Sink example, roughly 85 percent of pivots have a unique successor, and almost all the remaining 15 percent have only two branches. In the 3VdP example, more than 86 percent of pivots have a clearly unique branch; about 12 percent

of pivots have two branches; less than 1 percent of pivots have three branches; about 0.5 percent of pivots have four branches; and it is rarely that a pivot has more than four branches.

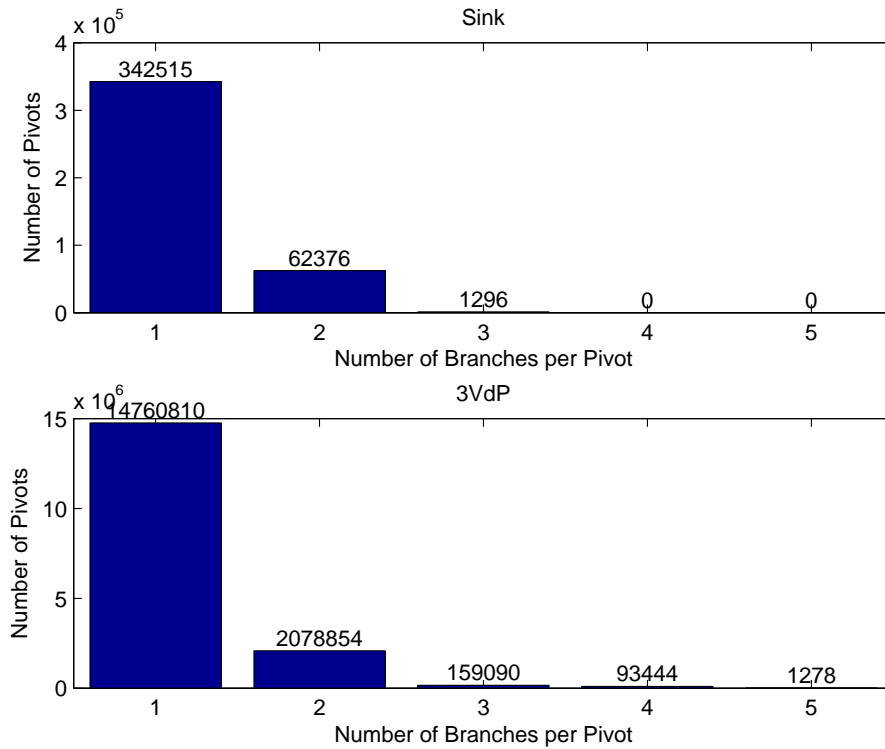


Figure 6.5: Distribution Number of Branches of a Pivot

The analysis of ill-conditioned system exceptions shows the necessity of the handling with badly conditioned problems. There are 8692 ill-conditioned linear systems found in the Sink example, about 0.01 per linear program. The 3VdP is more complex and therefore has more ill-conditioned system exceptions. There are about 0.75 million exceptions totally, which is 0.36 per linear program. The experimental data shows our estimate of the condition number to be quite effective. All exceptions in the Sink example and more than 99.7 percent of exceptions in the 3VdP example are caught by the estimated condition number. The method only

fails for less than 0.3 percent in the complex 3VdP example. Most of the badly conditioned linear systems have huge condition number; more than 81 percent have condition numbers greater than 10^{16} . The distribution is shown in Figure 6.6.

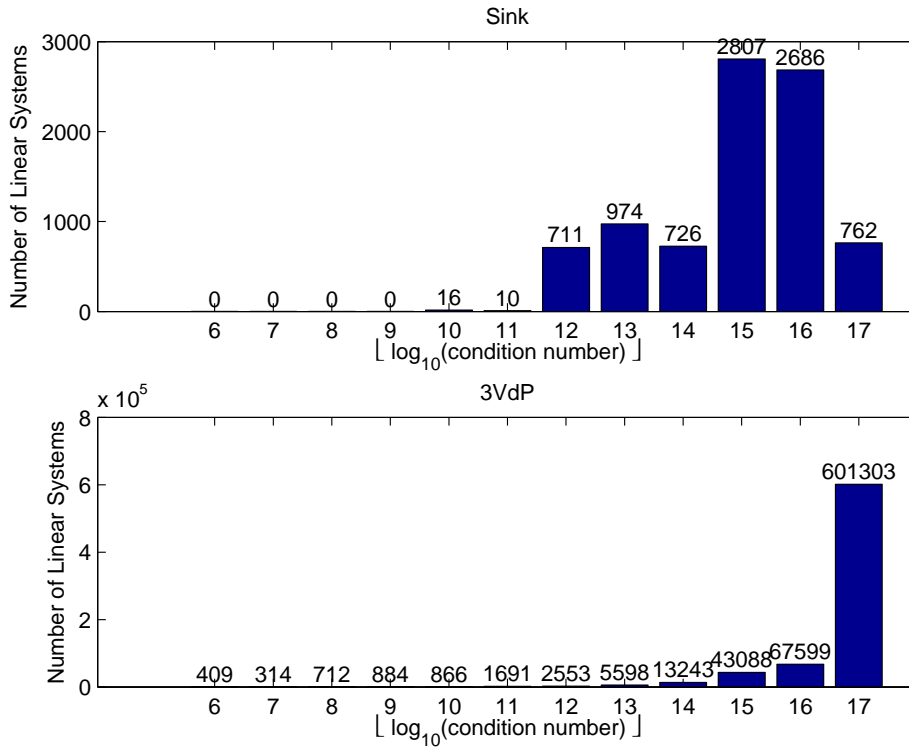


Figure 6.6: Distribution of Condition Number for Ill-Conditioned Problems

As described in Chapter 4.5, we tried two methods for finding an initial feasible basis. The quick algorithm is performed first, and the “big M” method is used if the quick algorithm fails. However, the experimental data shows the percentage of linear programs for which the quick algorithm succeeds is quite low. There are only 2 percent of linear programs of 3VdP example that contain a simple feasible basis; while it is a little more than 0.2 percent in the Sink example. It is reasonable that the 3VdP example has more simple feasible bases because has many more constraints than the Sink example; therefore, the probability to find a

simple feasible basis is much higher. For higher dimensional models, the method might show its advantages. However, for the examples considered here, there is no significant advantage to including the “quick” method.

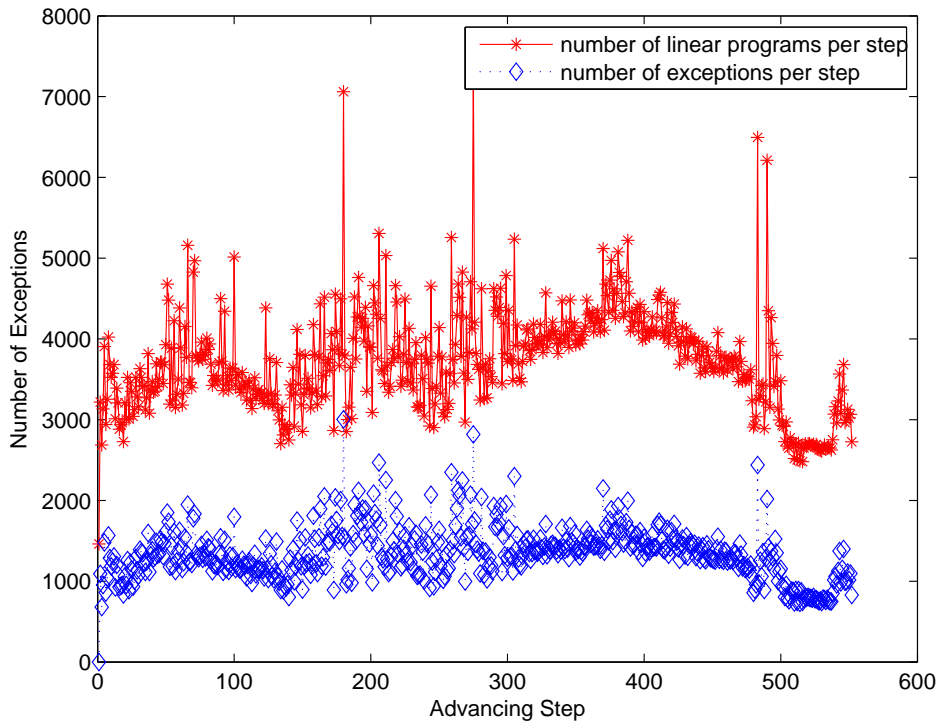


Figure 6.7: Number of Linear Programs and Exceptions per Step

Typically, the designer specifies a simple projectagon such as a hypercube for the initial reachable set. As reachability computation progresses, COHO produces more complicated projectagons to approximate the curvature of the actual trajectories. If the number of polygon vertices grows at each step, then COHO would become impractically slow.

The analysis of the number of linear programs and exceptions along advancing steps shows that COHO limits the complexity of the projectagons quite successfully. As shown in Figure 6.7, the number of linear programs oscillates around a

constant value, which shows the computation complexity for each step remains relatively constant. However, there are some steps that have a much larger number of linear programs. This is caused by the projection algorithm, as described in Chapter 5.2. Backtracking is performed when the algorithm jumps to the next quadrant incorrectly. In rare cases, many linear programs must be solved to compute the correct result. We should improve the projection algorithm in the future. The number of exceptions has similar trends, thus the number of exceptions per linear programs almost remains nearly constant. With a working version of COHO, we now have the opportunity to examine a large sample of the ill-conditioned problems that arise in COHO's analysis. This should allow us to test various conjectures as to their cause and may enable a more efficient solution to these problems in the future.

In summary, the experimental results have shown the efficiency and robustness of our COHO implementation.

Chapter 7

Conclusion

7.1 What has been Accomplished

We have implemented a new linear program solver for use in the COHO verification system and demonstrated that it is efficient, accurate and robust. The framework of this solver was set-up by Laza, who explored the special structure of COHO linear programs, presented an efficient $O(n)$ linear system solver and combined it with the Simplex algorithm. We implemented and improved his linear system solver and integrated it with interval arithmetic which guarantees the over approximation as required by verification. The algorithm to handle badly conditioned problems finds a good approximation of the optimal result and greatly improves the accuracy of the linear program solver. The experimental results demonstrate the accuracy, efficiency and robustness of our solver.

The projection problem, computing the projection of a projectagon onto a two-dimensional subspace, is solved by our projection algorithm using the COHO linear program solver. The algorithm produces a projected polygon that might be a slight over approximation. We derived an analytical upper bound for this error and showed from our experiments that it is negligible in practice.

We re-implemented the portions of COHO that rely on linear programming to use our new algorithms for solving linear programs and projecting the feasible region

onto two-dimensional subspaces. The new version of COHO has been applied to two practical examples. The success of these verification demonstrates the correctness of the ideas of COHO system such as projectagon representation of reachable region, approximation of the model, etc. The experimental data from these examples further demonstrate the soundness of the linear program solver and projection algorithm; otherwise, the error from these functions would have prevented COHO from verifying these complex examples.

In summary, this research has solved the previously known problems of COHO linear programs and projection of a projectagon; furthermore, the implementation has been shown to be efficient and accurate enough for practical verification tasks. In the process, the code base for COHO has been extensively revised and re-organized to facilitate further enhancements. We sketch some of these possibilities in the next section.

7.2 Suggestions for Further Research

The study of COHO linear programs and projection of the projectagon is not yet complete. There are many problems to explore further. The COHO verification tool has just begun to show its ability for verifying real circuits and other designs; this has opened up many areas for further research.

Our implementation of interval arithmetic as described in Chapter 3.2 over approximates the error bound. The IEEE floating point standard [Gol91] defines the relative error of floating point computation to be less than $\frac{1}{2}ulp$. However, COHO overapproximates it by a full ulp . We could get a tighter bound and improve the speed of the software by using specialized rounding modes that are included in the IEEE standard. In particular, the IEEE standard defines several rounding modes including round-up and round-down. The upper bound and lower bound can be obtained by these two rounding modes. However, the Java language does not support it. Thus, C or assembly language would be needed to access these

features, and JNI (the “Java Native Interface”) could be used to integrate this within our Java packages. By implementing interval versions of some basic linear algebra operations such as vector dot product, we could avoid the overhead of large numbers of calls between Java and C and realize much of the performance of a fully optimized implementation.

The optimal bases returned by COHO linear program solver are only numerically possible optimal; as described in Chapter 4.2.2, the bases returned by the solver can include infeasible, super-optimal bases as well. The inclusion of extra bases is caused by the estimation of computation error from interval arithmetic; it could be avoided if there is no round off error, i.e., the number has arbitrary precision. Arbitrary precision rational (APR) arithmetic is an alternative to interval arithmetic or floating point computation in which all computations are exact. Using APR, we could implement a linear program solver that would only return the optimal basis (or bases if the cost function is orthogonal to a critical constraint). Using APR, most of the issues described in Chapter 4.2 would be automatically addressed. We can implement an arbitrary precision arithmetic package based on the `BigInteger` class, or use a mathematical library like GMP [The].

The disadvantage of APR computation is that it is not directly supported by standard hardware. Thus, it will run slower than the corresponding floating point or interval number methods. Furthermore, APR methods tend to use more memory, and for general computation, this the size of an APR tends to grow with the length of the computation. Fortunately, our implementation of Simplex starts from the original data after each pivot and should therefore avoid much of this memory overhead. Furthermore, a hybrid algorithm could be used to improve efficiency: the interval method would be applied to find a small set of possibly optimal bases in the Simplex algorithm, the arbitrary precision rational arithmetic would only be used at the last step to identify the truly optimal basis from this set. The optimality of a basis can be easily checked by computing its feasibility for primal and dual linear

programs. Therefore, this algorithm should have efficiency that is comparable to that of the purely interval arithmetic method.

With arbitrary precision rational representation, computation is error free even for badly ill-conditioned problems. Therefore ill-conditioned basis should no longer be a problem. For the hybrid method, once an ill-conditioned problem is encountered, the solver could be restarted using APR methods for the particular problem. Therefore, the optimal basis is always available, and it is efficient because the badly conditioned problems rarely happen. We don not have to worry about the optimal basis being truly singular. This would mean that some constraint is exactly implied by some other set of constraints, and the singular basis describes a line. Because the feasible space is bounded by construction, there are other constraints that impose endpoints on the feasible portion of this line. A basis that replaces the redundant constraint with one of these endpoint constraints is both optimal and non-singular.

For the error of our linear program solver, we believe that the maximum error in the cost is bounded by the diameter of the reachable regions times the square root of the machine precision. This conjecture is supported by our numerical data as shown in Chapter 6.2, but we have not completed a formal proof. If APR methods are used, the exact optimal basis can be found; therefore, the relative error in the optimal cost will be due to the use of double precision numbers to describe the original linear program and to report the final result. Thus, the error will be determined by the machine precision.

Another idea to improve efficiency is to accelerate the pivot selection method. Consider how simplex does pivot selection: for each column j , the marginal cost \bar{c}_j is computed by equation 4.6. Combining this with equation 4.5, we obtain:

$$\bar{c}_j = c_j - c_{\mathcal{B}}^T \cdot (A_{\mathcal{B}}^{-1} A_{:,j}) \tag{7.1}$$

Computing t_j by equation 4.5 takes $O(n)$ operations using our special linear system solver. Likewise, calculating \bar{c}_j takes $O(n)$ operations. Thus this step takes $O(n)$

operations per column considered.

Matrix multiplication is associative. Therefore, equation 7.1 can be rewritten as:

$$\bar{c}_j = c_j - (c_B^T A_B^{-1}) \cdot A_{:,j} \quad (7.2)$$

Now, we can compute $(c_B^T A_B^{-1})$ using our $O(n)$ linear system solver. This is the same for all columns considered so we only need to calculate it once. In other words, we first compute

$$d = (c_B^T A_B^{-1}) \quad (7.3)$$

and use it for each column as

$$\bar{c}_j = c_j - d \cdot A_{:,j} \quad (7.4)$$

Here, we can take advantage of the sparsity of A. Any given column of A has either one or two non-zero elements. Thus, $d \cdot A_{:,j}$ can be calculated in $O(1)$ time.

Using this method, the computation time for a pivot is reduced from $O(m(n-m))$ to $O(n+m)$, where m is the number of constraints and n is the number of variables.

In the current version of the projection algorithm, all linear programs are treated as independent optimization problems by the COHO linear program solver. However, the sequence of linear programs to find the vertices of the projection polygon has a special structure. The optimal basis of such a linear program corresponds to a vertex of the polygon. The optimal bases of adjacent linear programs correspond to the two endpoints of an edge. Therefore, these two optimal bases is only a single pivot away. The COHO linear program solver should be able to exploit this special structure and improve the efficiency of projection algorithm greatly.

For the COHO system, obviously, this is just a start. There are many topics to study in the future. The top priority is to apply it on more examples, especially non-linear circuits for high-speed digital applications. We are also interested in examples from hybrid systems and control as well as models of biological process.

Projectagons provide an attractive way to represent high dimensional objects. Projectagons can represent non-convex objects, and operations on projectagons are efficient in terms of both time and space. Exploring the issues of projectagon geometry is another area for future work. For example, the intersection of two or more projectagons is obtained by the intersection of their projection polygons. On the other hand, projectagons are closed under neither complement nor union, and we do not know how to calculate the smallest overapproximation of the union. Such operations would be useful for implementing frontier based methods for reachability that could offer significant improvements of efficiency. Likewise, we do not know of an efficient algorithm for determining whether or not a projectagon is empty.

It appears that COHO is well-suited for a highly parallel implementation. Within each time step, the operations for each edge of each face are independent; accordingly, they could be computed concurrently. The amount of data output for each edge is small compared with the amount of work done to advance it through a time step. Therefore, parallel computing or cluster computing techniques should be applicable for improving the speed of COHO verification.

The procedure for linearizing the non-linear ODE model as described in Chapter 2.2 is done manually for each individual examples. This is too cumbersome for use by practicing hardware designers or control engineers. The ability to approximate the model automatically should be added in a future version. We are presently working on this for circuit verification applications. Once we have manually constructed models for transistors, we believe that these models can be composed directly from the structure of the circuit to produce a model for the circuit. Thus, no further manual linearization should be needed once we complete a generic model for transistors.

Bibliography

- [ADM02] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid systems. In *Proceedings of the Fourteenth Conference on Computer Aided Verification*, pages 365–370, Copenhagen, July 2002. Springer.
- [BA] Elliot Joel Berk and C. Scott Ananian. JLex: A lexical analyzer generator for Java(TM). <http://www.cs.princeton.edu/~appel/modern/java/JLex/>.
- [Bla77] R.G. Bland. New finite pivoting rules for the simplex method. *Math. Oper. Res.*, 2:103–107, 1977.
- [BMP99] Olivier Bournez, Oded Maler, and Amir Pnueli. Orthogonal polyhedra: Representation and computation. In *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*, pages 46–60. Springer, 1999. LNCS 1569.
- [BO79] J.L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28:643–647, 1979.
- [BT00] O. Botchkarev and S. Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. *Hybrid Systems: Computation and Control*, pages 73–88, 2000. LNCS 1790.
- [Dan63] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [DM98] Thao Dang and Oded Maler. Reachability analysis via face lifting. In Thomas A. Henzinger and Shankar Sastry, editors, *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, pages 96–109, Berkeley, California, April 1998.

- [DMn05] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, 2005.
- [Eme97] E. Allen Emerson. Model checking and the mu-calculus. *Proceeding of the DIMACS Symposium on Descriptive Complexity and Finite Model*, 1997.
- [Fre05] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *Proceedings of the Fifth International Workshop on Hybrid Systems: Computation and Control*, pages 258–273. Springer-Verlag, 2005. LNCS 3414.
- [GM98] Mark R. Greenstreet and Ian Mitchell. Integrating projections. In Thomas A. Henzinger and Shankar Sastry, editors, *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, pages 159–174, Berkeley, California, April 1998.
- [GM99] Mark R. Greenstreet and Ian Mitchell. Reachability analysis using polygonal projections. In *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*, pages 103–116, Berg en Dal, The Netherlands, March 1999. Springer. LNCS 1569.
- [Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.
- [GV96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins, 3rd edition, 1996.
- [Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceeding of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 278–292, 1996.
- [HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal of Software Tools for Technology Transfer*, 1(1-2):110–122, December 1997.
- [HPWT01] T.A. Henzinger, Joerg Preussig, and Howard Wong-Toi. Some lessons from the HyTech experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892. IEEE Press, 2001.

- [HS74] Morris W. Hirsch and Stephen Smale. *Differential Equations, Dynamical Systems, and Linear Algebra*. Academic Press, San Diego, CA, 1974.
- [Laz01] Marius D. Laza. A robust linear program solver for projectahedra. Master's thesis, Department of Computer Science, University of British Columbia, Vancouver, BC, December 2001.
- [PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.
- [PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.
- [RLB01] J. Douglas Faires Richard L. Burden. *Numerical Analysis*. Seventh edition edition, 2001.
- [SHA] Frank Flannery Scott Hudson and C. Scott Ananian. CUP: LALR parser generator for Java. <http://www.cs.princeton.edu/~appel/modern/java/CUP/>.
- [SK00] B.I. Silva and B.H. Krogh. Formal verification of hybrid system using Check-Mate: A case study. In *American Control Conference*, volume 3, pages 1679–1683, June 2000.
- [SK03] Olaf Stursberg and Bruce H. Krogh. Efficient representation and computation of reachable sets for hybrid systems. In *Proceedings of the Sixth International Workshop on Hybrid Systems: Computation and Control*, pages 482–497. Springer, 2003.
- [SR⁺00] B.I. Silva, K. Richeson, et al. Modeling and verifying hybrid dynamical systems using *checkmate*. In *Proceedings of the 4th International Conference on Automation of Mixed Processes (ADPM 2000)*, pages 323–328, 2000.
- [The] The GNU Project. GMP: Arithmetic without limitations. <http://www.swox.com/gmp/>.
- [Van01] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, second edition, 2001.
- [Win87] Wayne L. Winston. *Operations Research: Applications and Algorithms*. Indiana University, third edition, 1987.