

Topics in Cylindrical Algebraic Decomposition

submitted by

Corin Lee

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Mathematical Sciences

November 2025

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with the author. A copy of this thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with the author and that they must not copy it or use material from it except as permitted by law or with the consent of the author.

This thesis may be made available for consultation
within the University Library and may be
photocopied or lent to other libraries for the purposes
of consultation with effect from.....(date)

Signed on behalf of the Faculty of Science

Summary

This thesis presents several contributions designed to improve the practical application of cylindrical algebraic decomposition (CAD) by broadening its range of uses and improving accessibility. CAD is an important algorithm in symbolic computation with a variety of implementations and applications, yet its high computational complexity can be intimidating and means it should be used with care. However, this also means that seemingly small improvements and specialised methods can lead to significant performance improvements.

We present various approaches which help push CAD further in terms of reach, understanding, and application, starting with a detailed yet approachable overview. This is followed by demonstrating how a variety of novel tools and methods can be used to reduce the complexity of a problem, and showing how deeper consideration of problem features can make a significant difference. Additionally, we present the first implementation of CAD in *Macaulay2*, incorporating recent research and presenting CAD to a wider audience in a practical context. Further, we benchmark two contrasting CAD implementations in *Maple*, identifying scenarios in which one may be favourable. Finally, we introduce a graphing tool for CAD visualisation, with applications in both communication and research.

These contributions offer new ways to improve CAD and its accessibility, making it more approachable, transparent, and comparable for prospective users and researchers alike.

Acknowledgements

I would like to start by thanking Thomas Standen. This would not have been possible without your support. You encouraged me to apply, helped with my application, and believed in me when I did not believe in myself. You motivated me to continue through the most difficult periods and took the time to read, listen and understand a subject that is not particularly easy. I have grown a great deal during this time, and I have you to thank for helping me get started on this journey.

I would like to thank my supervisors, Professor Gregory Sankaran and Professor James Davenport, for their guidance throughout my time at Bath. Gregory, thank you for your support during challenging periods, your willingness to read my work in detail, and your thoughtful feedback. James, thank you for your continued enthusiasm and passion for the subject and my contributions, even after many years in the field. Your encouragement has helped me believe in myself and my work. I am also grateful for your constant feedback and willingness to help refine my drafts into something coherent, and for going above and beyond as a supervisor, providing many opportunities I would not have had otherwise.

I would also like to thank Dr. Russell Bradford and Dr. Pranav Singh for reading and assessing my confirmation report; your feedback helped guide my research. I extend my thanks to colleagues and researchers in the field, especially Ali Uncu, Matthew England, Tereso del Río, and Scott McCallum, for their encouragement, discussions, and inspiration over the years.

Special thanks go to everyone who helped me on the path to producing a package for *Macaulay2*. My deepest gratitude goes to Michael Stillman for co-authoring the software, suggesting that CAD would be worth implementing, and helping to bring the package together. I would also like to thank the University of Warwick for hosting the 2023 Computational Algebraic Geometry Workshop, where the package began, and Diane Maclagan, Alexander Kasprzyk, and Yue Ren for organising it. I would naturally

like to thank Tereso del Río and Hamid Rahkooy for co-authoring the package and their collaboration both during the workshop and long after it.

I would also like to thank Professor Elizabeth Mansfield, whose enthusiasm and passion inspired me during my Master's degree at the University of Kent and helped keep the spark alive after all this time.

I wish to thank all those who supported me during my time at the University of Bath. In particular, I thank Emma Finch and Trish Hunt, who went far beyond what their roles required, offered support when it was needed most, and were willing to delve into a topic others would not have.

I would like to thank the Doctoral College for hosting workshops, events, and writing retreats which provided a space to focus and meet with the wider doctoral community. I am also grateful for the opportunity to work as a Doctoral Development Facilitator, which allowed me to develop my professional and interpersonal skills.

My sincere thanks go to the University of Bath Library, particularly Research Services, for taking me on as part of the team during my studies. Thank you for giving me a sense of purpose, a source of income, and an environment in which I could demonstrate my skills. I am especially grateful to Junjie Shen, Sarah Ormes, and Catherine Borwick for working closely with me and helping me recognise my skills and strengths at times when the PhD work felt endless.

I want to extend heartfelt thanks to my friends and family for their support, encouragement, and companionship during this challenging period. To my mother, father, and grandmother: thank you for supporting me through every part of this journey despite any difficulties, and for helping with everything, big or small. To my extended family, thank you for your continued inclusion and accommodation. To Prince, Nathan, Basil, and Morgan: thank you for welcoming me home with open arms each time I returned, and for making the distance feel smaller. I love you all dearly and am so excited to see you again.

To Mahmoud, thank you for being the only other person who truly understood the challenges of this journey, and for being a confidant and companion throughout this journey.

I would like to extend special thanks to Timothy Massie. Your support and infectious enthusiasm were key in helping me persevere when it was hard to feel enthusiastic. Your belief in me helped me to believe in myself, and reminded me that my subject can be exciting, that my work has value, and that there is joy to be found in what I

do. You encouraged me to take pride in my work, and to do it with a smile. Your help has meant more than I can put into words. Thank you.

Finally, I thank everyone who took the time to read and review my thesis, especially my mother, Tom, Tim, Trish, and Abi. To everyone who asked about my work, listened with enthusiasm, and believed in me: thank you. Your kindness has made a difference.

Contents

1	Introduction	13
1.1	Objectives and Scope	13
1.2	Thesis Structure	14
1.3	Author Contributions and Collaborations	16
2	Background	17
2.1	Timeline	17
2.2	Applications	18
2.3	Quantifier Elimination	18
2.4	Multivariate Polynomials	21
2.4.1	Variable Ordering	21
2.4.2	Polynomial Terminology	22
2.5	Cylindrical Algebraic Decomposition	23
2.5.1	Projection and Lifting	26
2.5.2	Invariance and Delineability	27
2.5.3	The Sylvester Matrix	32
2.5.4	Basis, Primitive Part, Content, Squarefree	33
2.6	Projection and Lifting CAD Algorithms	36
2.6.1	Projection Operators, Lifting and Invariance	36
2.7	Variable Ordering Heuristics	38
2.8	Formula-specific Improvements	40
2.8.1	Truth-Invariance	40
2.8.2	Partial CAD	42
2.8.3	Block Ordering and Block Cylindricity	43
2.8.4	Open CAD	43
2.8.5	Equational Constraints	44
2.8.6	Truth-Table-Invariance	48
2.9	CAD via Regular Chains	50
2.9.1	Triangular Sets and Regular Chains	51

2.9.2	Complex Equivalents	51
2.9.3	Complex Cylindrical Trees	53
2.9.4	The Incremental Algorithm	54
2.9.5	Regular Chains with Equational Constraints	57
2.9.6	Regular Chains and Truth-Table-Invariance	59
3	A Case Study: The Solotareff Approximation Problem	62
3.1	Introduction	62
3.2	A Toolbox of Strategies	63
3.2.1	Leveraging Equations and Inequalities	63
3.2.2	Buchberger-Type Elimination Using Equational Constraints	64
3.2.3	Main Tools	66
3.2.4	Block Cylindricity	66
3.2.5	Symmetric Polynomials	68
3.2.6	Effective Quantification	69
3.2.7	Takeaways	69
3.3	Chebyshev's Theorem	70
3.3.1	Collins' Formulation	73
3.3.2	Parity Simplification	74
3.3.3	Exploiting Inequalities	75
3.3.4	Further Simplifications	76
3.3.5	Reconsidering the Variable Ordering	78
3.3.6	General Elimination Strategies	80
3.4	Solutions for $n = 5$	81
3.4.1	Applying the Tools	82
3.4.2	Exploiting Symmetry	85
4	CAD in <i>Macaulay2</i>	89
4.1	Introduction	89
4.1.1	Motivating Example	90
4.2	Design Choices for the package	91
4.2.1	Open CAD	91
4.2.2	Improvements to <i>RealRoots</i>	91
4.3	The <i>CylindricalAlgebraicDecomposition</i> Package	92
4.3.1	Overview	93
4.3.2	Projection Phase	98
4.3.3	Lifting Phase	99
4.4	Performance of CAD on Intersecting Hyperspheres	102

5	Benchmarking CAD Algorithms in <i>Maple</i>	103
5.1	Introduction and Purpose	103
5.2	Background and Implementation	104
5.2.1	Heuristics	106
5.2.2	Comparison	107
5.2.3	Issues and Solutions	108
5.3	Setup and Layout	110
5.4	The Dataset	111
5.5	The Tests and Results	112
5.6	Results and Observations	113
5.6.1	Lex Ordering, Polynomials Only	114
5.6.2	Heuristic Ordering, Polynomials Only	115
5.6.3	Degree Sum Ordering, Polynomials Only	115
5.6.4	Lex Ordering, Exploiting the EC	116
5.6.5	Heuristic Ordering, Exploiting the EC	117
5.6.6	Degree Sum Ordering, Exploiting the EC	118
5.6.7	General Observations	120
5.7	Errors	124
6	GraphCAD - A <i>Maple</i> Tool for Visualising 2D CADs	125
6.1	Introduction	125
6.2	GraphCAD Procedures	127
6.2.1	Pre-processing Commands	127
6.2.2	CAD Construction and Graphing Commands	128
6.3	Choices and Limitations	130
6.4	Example Usage in <i>Maple</i>	132
7	Future Work	139
7.1	Further Focus on Regular Chains	139
7.2	Strategies for Effective Use of CAD	140
7.2.1	Formula and Ordering Considerations	140
7.2.2	Other Improvements and Heuristics	141
7.3	CylindricalAlgebraicDecomposition Improvements	141
7.4	Future Benchmarking Studies	142
7.5	Future Improvements for GraphCAD	142

List of Theorems, Propositions and Corollaries

2.3.1	Proposition (Real Quantifier Elimination)	19
2.3.5	Theorem (Quantifier Elimination)	20
2.3.6	Theorem (Tarski)	20
2.5.6	Theorem (Tarski-Seidenberg)	26
2.5.10	Theorem (Delineability and Real Root Structure)	30
2.5.12	Theorem (Main CAD Theorem)	32
2.5.16	Theorem (Subresultants and GCD)	33
2.5.17	Theorem (GCD and Distinct Roots)	33
2.5.18	Corollary (Resultants, Discriminants, and Roots)	33
2.5.23	Corollary (Resultants, Discriminants, Coprime, and Squarefree)	35
2.6.4	Theorem (General Lifting Theorem)	37
2.8.5	Theorem (Restricted Lifting Theorem)	46
2.8.6	Theorem (Semi-restricted Lifting Theorem)	47
2.8.9	Theorem (Reduced Lifting Theorem)	50
2.9.12	Theorem (Subresultants and GCD mod Γ)	54
3.3.1	Theorem (Chebyshev's Theorem, special case)	70

List of Definitions

2.3.2	Definition (Atomic formula, polynomial constraint)	19
2.3.4	Definition (Tarski formula, quantified formula, prenex normal form)	19
2.5.1	Definition (Cylinder, stack, section, sector)	23
2.5.2	Definition (Cylindrical decomposition)	24
2.5.3	Definition (Semi-algebraic set)	24
2.5.4	Definition (Cylindrical Algebraic Decomposition)	24
2.5.5	Definition (Sign-invariant)	25
2.5.7	Definition (Root structure, root structure matrix)	28
2.5.8	Definition (Delineable)	29
2.5.9	Definition (Delineable [Col98])	29
2.5.13	Definition (Sylvester Matrix)	32
2.5.14	Definition (Resultant, discriminant)	32
2.5.15	Definition (Subresultant, PSC)	33
2.5.19	Definition (Content, primitive part)	33
2.5.21	Definition (Basis)	34
2.5.22	Definition (Squarefree)	35
2.6.1	Definition (General projection operator)	36
2.6.2	Definition (Valuation)	37
2.6.3	Definition (Valuation-invariant)	37
2.7.1	Definition (Brown heuristic)	38
2.7.2	Definition (sotd heuristic)	38
2.7.3	Definition (mods heuristic)	39
2.8.1	Definition (Truth-invariance)	40
2.8.3	Definition (Solution set)	41
2.8.4	Definition (Equational constraint)	45
2.8.7	Definition (Truth-table-invariant)	49
2.8.8	Definition (Reduced projection (TTICAD))	49
2.9.2	Definition (Triangular set)	51
2.9.3	Definition (Regular chain)	51
2.9.4	Definition (Squarefree triangular set)	51

2.9.5	Definition (Simple system)	51
2.9.6	Definition (Separation)	51
2.9.7	Definition (Invariance)	52
2.9.8	Definition (Cylindrical decomposition of $\mathbb{C}^n$ and associated tree)	52
2.9.9	Definition (Solution set over $\mathbb{C}$)	53
2.9.10	Definition (Invertible/zero/coprime modulo Γ)	53
2.9.11	Definition (GCD mod Γ)	54
2.9.13	Definition (Complex system)	57
2.9.16	Definition (Truth-invariance on Γ)	59
2.9.17	Definition (Truth-table-invariant CCT/CCD)	60
5.2.1	Definition (Tonks heuristic)	107

List of Tables

3.1	The different forms of $\hat{E}_n(u_p)$ and $\hat{E}_n(u_q)$, depending on the parity of n .	78
3.2	Variable elimination strategies for different values of n .	80
5.1	Configuration options for each CAD command.	113
5.2	A table comparing the number of times one ordering outperformed the other for each algorithm, with the cumulative time difference across those cases.	119
6.1	The colours used to mark 2D cells.	130

List of Figures

2-1	$\mathcal{F}$ -sign-invariant CADs for $\mathcal{F} = \{x^5 - 5x^3 + 4x - y\}$. Using ordering $y \prec x$, we obtain a CAD with 59 cells (18 shaded areas, 29 curve segments and 12 points). Using the ordering $x \prec y$ generates only 3 cells (2 areas and one curve).	22
2-2	The 1D cell C and the 2D cylinder $\mathcal{Z}(C)$ above it.	24
2-3	The graph of $x^2 + y^2 - 1$ and the $\{x^2 + y^2 - 1\}$ -invariant CAD of $\mathbb{R}^2$	25
2-4	Outline of the projection and lifting method for constructing an $\mathcal{F}_n$ -invariant CAD of $\mathbb{R}^n$	27
2-5	The roots of $f(\mathbf{0}, x_n)$, ordered according to their positions on a general real line though the origin.	30
2-6	An overview of the incremental algorithm. Note that the steps on the left increment by polynomial, while those on the right increment by dimension, meaning each CCD on the left has no direct relation to the CAD to its right.	55
2-7	A CCD of $\{x^2 + y^2 - 1, x^3 - y^2\}$ and the resulting CAD, for the variable ordering $x \prec y$. Here α is the real root of $x^3 + x^2 - 1 = 0$	56
3-1	The graph of the lower form of $E_5(x)$, for different values of r . The green lines are $\pm L$	71
3-2	A rough sketch of the higher form of $E_n(x)$ for $r > S_n$, when n is odd (left) and when n is even (right).	72
3-3	Plot of the function $r^2 - 10r + 5$	82
4-1	Algebraic varieties of $3 - x^2$ (blue) and $(7x - 12)(x^2 + x + 1)$ (orange), and the associated values of x ensuring these polynomials are positive (blue and orange highlighting, respectively).	90
4-2	List of functions contained in the <code>CylindricalAlgebraicDecomposition</code> package. Methods labelled in blue are from external packages.	92
4-3	The system of polynomials $\mathcal{F} = \{f_1, f_2\}$, used as an example throughout this section.	93

4-4	A visualisation of the open CAD of $\mathcal{F}$. The coloured regions represent the open cells, the green diamond the positive sample point, and the black crosses the remaining sample points from <code>openCAD</code> .	95
4-5	A representation of these sample points in a tree structure.	95
4-6	A plot of the algebraic variety of $\mathcal{F}_n$. Two intersecting spheres centered at (1,1,1) and (-1,-1,-1) respectively.	102
5-1	An overview of each method. Note that PL-CAD counts down dimensions, whereas RC-CAD counts down polynomials.	104
5-2	Cactus plot comparing the performance of PLCAD and RCCAD on collections of polynomials, using different variable orderings: lex, heuristic (Brown) and <code>dsum</code> .	116
5-3	Cactus plot comparing the performance of PLCAD and RCCAD on DNF formulae containing ECs, using different variable orderings: lex, heuristic (Brown for RCCAD, Tonks for PLCAD) and <code>dsum</code> .	119
5-4	Cactus plot comparing the performance of RCCAD on both polynomials and on DNF formulae containing ECs, using different variable orderings: lex, heuristic (Brown) and <code>dsum</code> .	121
5-5	Cactus plot comparing the performance of PLCAD on both polynomials and on DNF formulae containing ECs, using different variable orderings: lex, heuristic (Brown/Tonks) and <code>dsum</code> .	121
5-6	Cactus plot comparing each combination for PLCAD and RCCAD. Virtual best (VB) is plotted as a black line.	122
5-7	Cactus plot comparing the performance of PLCAD and RCCAD on DNF formulae containing ECs against PLCAD on formulae not converted to DNF (in green) using the three orderings.	123
6-1	List of <code>GraphCAD</code> commands. For simplicity, internal commands (grey) are left unexpanded, while commands from other packages are omitted.	127
6-2	CAD graphs of $x^3 - y^2$ using PL and RC respectively. Both have the projection ordering $x \prec y$, but different display orderings, with the 1D CAD below.	137
6-3	CAD graphs of $x^3 - y^2$ showing various combinations of projection and display order.	137
6-4	CAD graphs of $x^3 - y^2$ showing the 1D CAD below and sample points (left), and only open cells (right).	138
6-5	The 2D CAD associated to the 2D projection polynomials of $R1$.	138

Chapter 1

Introduction

This thesis presents a number of approaches designed to improve the practical application of cylindrical algebraic decomposition (CAD) by expanding its range of uses and improving accessibility. We begin with an extensive but non-comprehensive account of CAD, before presenting four of these approaches. The associated chapters are largely unconnected, can be read independently, and aim to be as clear and approachable as possible.

In the first of these chapters, we present a case study demonstrating the effectiveness of a variety of tools and methods in reducing the complexity of a problem. In the second chapter, we expand the reach of CAD to a wider user base through its first implementation in the open-source program *Macaulay2*. The remaining two chapters focus on implementations within *Maple*, presenting both a benchmarking comparison between two CAD algorithms and a graphing tool to aid with visualisation.

1.1 Objectives and Scope

Cylindrical algebraic decomposition is an important algorithm in symbolic computation for studying real semi-algebraic sets and as a method for real quantifier elimination (QE) [Col75]. CAD has been implemented in many widely used computer algebra systems and has applications across mathematics, science, and engineering. However, CAD is known to have worst-case doubly exponential complexity [DH88]. Despite this, many important cases and applications are within scope, and small algorithmic improvements and specialist methods lead to significant performance improvements.

This thesis looks at two ways to make CAD more usable: one is to expand the range of problems to which it can be applied by bringing them into scope, and the other is to improve accessibility by removing barriers to comprehension and use. Each chapter independently tackles one or both of these approaches, with the connecting thread of communication: CAD and QE have many more undiscovered applications, but their inherent complexity may intimidate potential users, and use without proper care can lead to poor performance. The aim of the thesis is to ensure clarity and approachability, with intuitive explanations accompanied by visual summaries and examples.

An important aspect of communication is knowing when CAD is appropriate. CAD is applicable to many areas, but its suitability and limitations are more difficult to determine *a priori*, even for specialists. A well-formulated setup can often be the deciding factor in whether a problem is tractable, and this may make CAD unappealing, and arguably harder to promote than if it simply failed to work for higher-dimensional or higher-degree problems.

We aim to improve communication by identifying special features that can be utilised to simplify CAD construction and make certain problem types more tractable. We also aim to improve the usability, practicality, and availability of CAD through additional implementations and communicative tools, allowing us to highlight areas of worthy future research.

In particular, we highlight two competing algorithm classes for CAD; the classical projection and lifting approach (PL-CAD), and the newer incremental regular chains approach (RC-CAD). Unfortunately there is no systematic side-by-side comparison of these in the literature, so Chapter 2 aims to fill this gap, to serve as a solid foundation for the rest of the thesis and, the author hopes, a useful resource in itself. These competing approaches are compared further in Chapter 5 and their outlines are illustrated in Figure 5-1.

1.2 Thesis Structure

Chapter 2 provides the background for the thesis, thoroughly outlining CAD and QE in a clear and concise manner, whilst also unifying the various methods and ideas developed over the last fifty years. This includes a timeline and applications, a unification of historic projection operators and notions of invariance and delineability, an account of block ordering and block invariance, and a comprehensive account of CAD via regular chains.

In Chapter 3, we introduce various tools and strategies and apply them to a suitable case study, demonstrating the power of identifying key features and formulating problems carefully in order to exploit them. Using the Solotareff problem as an example, we show how exploiting symmetry, choosing effective variable orderings, and eliminating free variables early can dramatically reduce the complexity of a problem, turning a previously intractable problem into one that could be solved by hand.

In Chapter 4, we present the first implementation of CAD in *Macaulay2*, extending CAD's reach to a wider audience. This implementation incorporates recent improvements, such as the Lazard projection and a new heuristic, while addressing inefficiencies discovered in a previous package.

Chapter 5 focuses on benchmarking two contrasting CAD implementations in *Maple*: a PL-CAD implementation, and an RC-CAD method that works incrementally. There are currently no clear indicators for when to use one over the other, so these experiments represent an important first step towards this, and offer some striking results, such as suggesting that a widely used variable ordering heuristic performs poorly in practice while a simpler one performs much better. The chapter provides an initial comparison of these two very different implementations, illustrating the strength of each, alongside various tools to enable fairer comparisons and facilitate future experiments.

Chapter 6 introduces **GraphCAD**, a *Maple* tool for constructing and visualising 2D CADs which interfaces with both PL-CAD and RC-CAD implementations. By unifying these vastly different implementations, **GraphCAD** aids in communication and exploration by enabling direct visual and structural comparisons, and provides an additional way to present CAD to a wider audience.

We then conclude with a section on future work, giving natural extensions to the ideas introduced in this thesis.

Though the means are different, these topics are united by a common aim: pushing CAD further in terms of reach, understanding, and application. Chapter 3 demonstrates promising application for various tools, many of which have not been used in any implementation to date. Chapter 4 introduces CAD to a new audience and implements recent research. Chapter 5 directly compares two rival CAD methods and implementations, bringing more attention to RC-CAD as a direction for future research. Chapter 6 improves communication and comparison through visualisation. Together, these make CAD more approachable, transparent and comparable, for new users and researchers alike.

1.3 Author Contributions and Collaborations

Chapter 4 is based on [LdR25], which describes work done in collaboration with Tereso del Río and Hamid Rahkooy. The author serves as the lead point of contact for the corresponding *Macaulay2* software [Lee25b], which has already attracted interest from several users.

The remainder of the thesis, including the **GraphCAD** software package for *Maple* [Lee25a], is the sole work of the author, supervised by Professor Gregory Sankaran and Professor James Davenport.

The tools, packages, and resources developed for and used in this thesis are available at <https://github.com/cel34-bath/CADRepository>.



CADRepository - GitHub

Chapter 2

Background

In this chapter, we provide the background on real quantifier elimination and cylindrical algebraic decomposition needed for the remainder of the thesis. We begin with a detailed overview of key features and concepts, before introducing several of the major improvements introduced throughout the development of CAD.

2.1 Timeline

Tarski [Tar51] proved the existence of a decision procedure for sentences in real closed fields and showed that real quantifier elimination was possible. Seidenberg [Sei54] showed that the projection of a semi-algebraic set is itself semi-algebraic, and this projection corresponds to the elimination of variables.

CAD was first introduced by Collins in 1975 [Col75] as a method and algorithm for performing real QE, with a first complete implementation by Arnon a few years later [Arn81]. An improved projection operator was given by McCallum [McC84], and Davenport and Heintz [DH88] established that QE, and therefore CAD, has worst-case doubly-exponential complexity. Further developments include the concept of partial CAD [CH91], improvements for CAD-based QE [Hon90, Hon92] and projection improvements. Hong [Hon90] demonstrated a subset of Collins' projection operator was sufficient, and McCallum showed a smaller projection operator, whose proof relied on order-invariance, works for three dimensions [McC88]. This was later generalised to n dimensions [McC98], but fails when nullification occurs during the lifting phase (in regions known as *curtains* [Nai21]). McCallum then introduced restricted projection operators for exploiting equational constraints [McC99, McC01]. A further improved projection operator was proposed in [Laz94], but its proof was flawed.

An alternative method of producing CADs via complex space using regular chains was produced, first a recursive method using triangular sets [CMMXY09], then an incremental algorithm via refining complex trees [CMM14a]. Truth-table-invariant CADs were introduced by Bradford et al. [BDE⁺13] for the original method, followed by a regular chains equivalent [BCD⁺14].

The flaw with the Lazard projection was detailed in [MH16], which showed the projection held only when curtains were not present, and the projection was finally validated in [MPP19], enabling its use in practice. Subsequent work included restricted projections [Nai21], first implemented in [Ton21], and a further refined projection operator [BM20, DNSU23].

Notable software that supports QE and CAD includes QEPCAD [Bro, CH91], *Redlog* [DSW98], *SyNRAC* [YA07] and *Mathematica* [Res], with more recent implementations including *SMT-RAT* [CKJ⁺15] and TARSKI [VEB18]. Of particular note are the implementations in *Maple* of both regular chains CAD algorithms [CMMXY09, CMM14a], and a projection and lifting algorithm using the Lazard projections and subsequent work [Ton21], which are key for Chapter 5 and 6 of this thesis.

2.2 Applications

QE and CAD have applications in many fields. They have been used in biochemical network analysis [BDE⁺17], epidemic modelling [BENW06] and to disprove a biological conjecture [RS21]. There have also been multiple applications in robotics [MMC⁺12, CSEDS20, SS83], applied mechanics [Ioa17a, Ioa17b, CC18], and economic reasoning [MDE18]. Further uses include artificial intelligence [WMTA16], derivation of optimal numerical schemes [EH16], parametric optimisation [FPM05], theorem proving [Pau12], reasoning with multivalued functions [DBEW12], and proving collision avoidance under cruise control [ST11]. More recently, CAD has been used in astrophysics to classify unobscured telescope designs [ADD25]. Across these examples, CAD often provides exact solutions where numerical methods might fail or introduce errors.

2.3 Quantifier Elimination

Quantifier elimination (QE) is the process of transforming a logical formula involving quantifiers into a logically equivalent formula without them. The study of QE in real closed fields originates with Tarski [Tar51], who proved the existence of a decision procedure for sentences in real closed fields, in particular a method for real QE.

Proposition 2.3.1 (Real Quantifier Elimination). *Let Φ be a formula in R . Then there exists a quantifier-free formula Ψ , with the same free variables as Φ , such that for every real closed extension K of R and every x in K^n , $\Phi(x) \Leftrightarrow \Psi(x)$.*

The following definitions are taken from [Col75, ACM84, Nai21, EBD20, Jir95, Tar51].

Definition 2.3.2 (Atomic formula, polynomial constraint). *An atomic formula is an expression of the form*

$$f \diamond 0, \quad \diamond \in \{=, >\}, \quad (2.1)$$

in which $f \diamond 0$ is the polynomial constraint and $f \in \mathbb{R}[x_1, \dots, x_n]$ is the constrained polynomial¹. Of particular note are expression in the form $f = 0$, which are referred to as equational constraints (see Section 2.8.5).

Remark 2.3.3. *When working with QE and CAD, there are three main fields of interest. The underlying abstract theory holds for any real closed field R , usually taken to be the reals $\mathbb{R}$, as we are concerned with real roots and real QE. For the purposes of computation, however, one works with polynomials with rational (or equivalently integer) coefficients, whose roots are algebraic numbers*

$$\mathbb{A} = \overline{\mathbb{Q}} \cap \mathbb{R} = \{a \mid f(a) = 0 \text{ for some } f \in \mathbb{Q}[x]\}.$$

The field $\mathbb{A}$ is computable, unlike $\mathbb{R}$, and is needed for full CAD construction².

This is not always clearly emphasised: in the literature (and indeed in this thesis) it is common to see in the context of real QE and CAD of $\mathbb{R}^n$ the notation $f \in \mathbb{R}[x_1, \dots, x_n]$ used to denote a polynomial with rational coefficients, with the intention that whenever the variables are fixed, they are only ever evaluated at algebraic values of the x_i , so that any polynomials that arise this way have algebraic coefficients, and if they are univariate, algebraic roots. While this notation is not wrong (since $\mathbb{Q} \subset \mathbb{A} \subset \mathbb{R}$), clarification is useful when discussing implementation.

Definition 2.3.4 (Tarski formula, quantified formula, prenex normal form). *A (Tarski) formula is a finite collection of atomic formulae connected by the standard Boolean operators $\neg, \wedge, \vee$, whose variables may be quantified by $\exists$ or $\forall$ over the reals. Such a formula is called quantified if it contains quantifiers, and quantifier-free otherwise.*

¹Is it common to also see $\geq, <, \leq$ and $\neq$ used here. While these are not necessary, as polynomials with these constraints can be produced by Boolean combinations of atomic formulae, they are useful shorthand.

²It is still possible to construct an open CAD when restricted to $\mathbb{Q}$ (see Chapter 4), but a method for symbolically storing and tracking roots (such as *Maple*'s `RootOf` structure) is needed for any implementation of the full CAD algorithm.

Any quantified formula can be written in prenex normal form:

$$\Phi = Q_{k+1}x_{k+1} \dots Q_n x_n \phi(x_1, \dots, x_n), \quad (2.2)$$

where each $Q_i \in \{\exists, \forall\}$. In this situation, $Q_{k+1}x_{k+1} \dots Q_n x_n$ is called the quantifier prefix and ϕ is quantifier-free. Variables in the quantifier prefix are quantified variables, and the remaining variables are free variables. A consecutive sequence of quantified variables in the quantifier prefix bound by the same quantifier is known as a quantifier block.

For the remainder of this thesis, all quantified formulas are assumed to be in prenex normal form. We now restate quantifier elimination:

Theorem 2.3.5 (Quantifier Elimination). *Let Φ be a quantified formula in the form (2.2). Then there exists a quantifier-free formula $\Psi(x_1, \dots, x_k)$ that is logically equivalent, i.e., $(\forall x_1, \dots, x_k) (\Phi \leftrightarrow \Psi)$,*

Additionally, any such quantifier-free formula can be expressed in *disjunctive normal form* (DNF):

$$\bigvee_i \bigwedge_j (f_{i,j} \diamond_{i,j} 0). \quad (2.3)$$

Theorem 2.3.6 (Tarski). *Every Tarski formula Φ is decidable: its truth value can be determined in finitely many steps. There is a decision method for the class of all true sentences of elementary algebra.*

A simple example of quantifier elimination is that of solving a non-trivial quadratic equation over the reals: the initial statement

$$(\exists x \in \mathbb{R}) (ax^2 + bx + c = 0 \wedge a \neq 0)$$

has the corresponding quantifier-free solution:

$$(a \neq 0) \wedge (b^2 - 4ac \geq 0),$$

where a, b, c are free variables.

One important method for QE is *cylindrical algebraic decomposition* (CAD). Given a set of polynomials in n variables, a CAD is a partitioning of $\mathbb{R}^n$ into finitely many disjoint regions known as cells, with some useful properties. In particular, each input polynomial has constant sign (positive, negative, or zero) on a cell; therefore it is sufficient to check a finite number of *sample points*, one for each cell, to determine all possible sign combinations.

CAD was the first method to make real QE algorithmically feasible, using a recursive *projection and lifting* algorithm; first computing successive projections to produce equations in fewer and fewer variables until a set of univariate polynomials is obtained. The CAD of $\mathbb{R}$ is the collection of each root of these polynomials and the regions either side of them. A sample point of each cell is then *lifted* into the polynomials in one variable greater, producing a new set of univariate polynomials and thus allowing the process to continue, producing a CAD of $\mathbb{R}^n$ composed of cells in which the original polynomials have constant sign.

For a quantifier-free Tarski formula, the solution set is simply the union of true cells (the disjunction of the formulae which define each cell), which can be determined by evaluating the sample points. For a quantified Tarski formula, repeated projection can be used to produce a set of polynomials in the free variables only. This is our quantifier-free equivalent formula, and the truth value of its cells can be found by propagating the truth values from the input formula. We shall focus on the former case over the next couple of sections, and will look further at the latter case in Section 2.8.1.

2.4 Multivariate Polynomials

2.4.1 Variable Ordering

An important aspect of QE and CAD is how multivariate polynomials are treated, in particular how the variables are ordered. CAD requires a declared ordering on the variables, and when used for real QE, the variables must be ordered to match the quantifier prefix in order to obtain the quantifier-free equivalent formula.

For a collection of variables $\mathbf{x} = (x_1, \dots, x_n)$ equipped with a total order $\prec$, we say $x_j \prec x_i$ if x_i is greater than x_j in this ordering. The ordering determines the sequence in which variables are projected during CAD. As is standard in the CAD literature, we assume in general the ordering $x_1 \prec \dots \prec x_n$.

The choice of variable ordering has a significant impact on cell count and thus performance [DSS04], with Brown and Davenport [BD07] giving an example in which CAD complexity can be constant or doubly exponential depending on the variable ordering. This has motivated research into variable ordering strategies, using problem features to produce heuristics, some of which are given in the following subsection, and more recent work using machine learning [EF19, HEW⁺19, PdEC24].

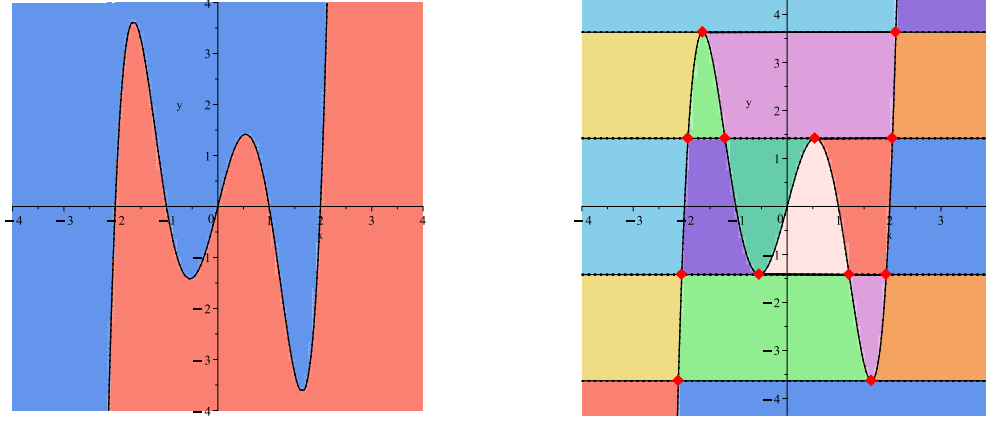


Figure 2-1: $\mathcal{F}$ -sign-invariant CADs for $\mathcal{F} = \{x^5 - 5x^3 + 4x - y\}$. Using ordering $y \prec x$, we obtain a CAD with 59 cells (18 shaded areas, 29 curve segments and 12 points). Using the ordering $x \prec y$ generates only 3 cells (2 areas and one curve).

Figure generated by **GraphCAD** (see Chapter 6).

2.4.2 Polynomial Terminology

For a polynomial $f \in \mathbb{R}[x_1, \dots, x_n]$, its *main variable*, denoted $\text{mvar}(f)$, is the greatest variable with respect to $\prec$ that appears in f . If $\text{mvar}(f) = x_k$, we say f is a polynomial of *level* k .

For the purposes of CAD, f is considered univariate in its main variable, with coefficients in the remaining variables. If we assume $x_1 \prec \dots \prec x_n$, we can write $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ as $f = \sum_{i=0}^d a_i x_n^i$, where each $a_i \in \mathbb{R}[x_1, \dots, x_{n-1}]$ and $a_d \neq 0$, and we define the following (with respect to x_n):

- The *degree* of f , $\deg(f) = d$.
- The *leading coefficient* of f , $\text{lc}(f) = a_d \in \mathbb{R}[x_1, \dots, x_{n-1}]$.
- The *leading term* of f , $\text{lt}(f) = \text{lc}(f)x_n^{\deg(f)} = a_d x_n^d$.
- The *trailing coefficient* of f , $\text{tc}(f) = a_0$ (the ‘constant term’ with respect to x_n).
- The *coefficients* of f , $\text{coeffs}(f) = \{a_d, \dots, a_0\}$.

In cases when a variable needs to be specified for these operators, it will be denoted in subscript. Otherwise, it is assumed these operators are taken with respect to $\text{mvar}(f)$.

When one needs to consider the degree of f in all variables, we define the *total degree* of f , $\text{tdeg}(f) = \max_j \{\sum_i \deg_{x_i}(m_j)\}$, where m_j are the monomials that make up f .

For a set of polynomials $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$,

- $\text{mvar}(\mathcal{F}) = \text{mvar}\{\text{mvar}(f_i)\}$,
- $\text{deg}(\mathcal{F}) = \max\{\text{deg}(f_i)\}$,
- $\text{lc}(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} \{\text{lc}(f)\}$,
- $\text{tc}(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} \{\text{tc}(f)\}$,
- $\text{tdeg}(\mathcal{F}) = \max\{\text{tdeg}(f_i)\}$.
- $\text{coeffs}(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} \text{coeffs}(f)$,

2.5 Cylindrical Algebraic Decomposition

Quantifier elimination over the reals was first achieved algorithmically through the use of *cylindrical algebraic decomposition* (CAD), introduced by Collins [Col75]. Such an algorithm takes set of input polynomials $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_n]$ and produces the namesake CAD, a *decomposition* of $\mathbb{R}^n$ into a finite collection of disjoint, non-empty connected subsets known as *cells*, whose union is $\mathbb{R}^n$, in which each polynomial in $\mathcal{F}$ is *sign-invariant*: its value is consistently positive, negative, or zero at every point in the cell. We call such cells *($\mathcal{F}$ -)sign-invariant*.

If presented with a quantifier-free Tarski formula ϕ in disjunctive normal form (2.3), the set $\mathcal{F}$ is simply the collection of each $f_{i,j}$, and one can check each sign-invariant cell of the CAD of $\mathcal{F}$ to determine when the formula is true or false, ensuring the input formula is also *truth-invariant* within each cell, and each cell is *(ϕ -)truth-invariant*. The notion of truth-invariance is discussed in more detail in Section 2.8.1.

We now formally define a CAD.

Definition 2.5.1 (Cylinder, stack, section, sector). *Given a cell $C \subseteq \mathbb{R}^{n-1}$, define the cylinder $\mathcal{Z}(C) = C \times \mathbb{R}$. Given $k \geq 0$ continuous real-valued functions $f_1 < \dots < f_k$ on C , a stack over C is the decomposition of $\mathcal{Z}(C)$ into the regions $(S_1, \dots, S_{2k+1})$, consisting of :*

- *f_i -sections $S_{2i} = \{(a, f_i(a)) \mid a \in C\}$ for $1 \leq i \leq k$,*
- *(f_i, f_{i+1}) -sectors $S_{2i+1} = \{(a, b) \mid a \in C, f_i(a) < b < f_{i+1}(a)\}$ for $0 \leq i \leq k$, where $f_0 = -\infty$ and $f_{k+1} = +\infty$.*

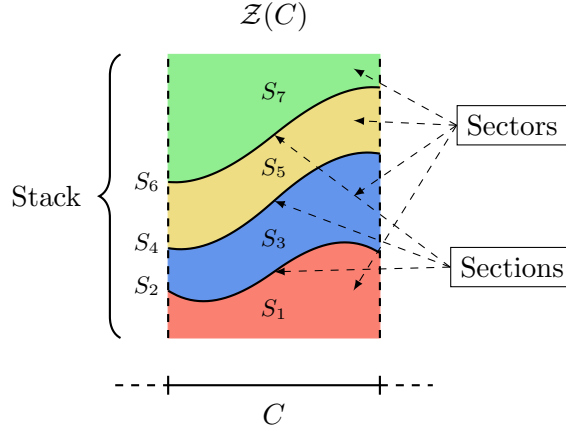


Figure 2-2: The 1D cell C and the 2D cylinder $\mathcal{Z}(C)$ above it.

A cell $C \subseteq \mathbb{R}^{n-1}$ is called an i -cell for $0 \leq i \leq n-1$ if it is homeomorphic to $\mathbb{R}^i$. If C is an i -cell, the sections and sectors of $\mathcal{Z}(C) \subseteq \mathbb{R}^n$ are i -cells and $(i+1)$ -cells, respectively.

Definition 2.5.2 (Cylindrical decomposition). *A decomposition $\mathcal{D}$ of $\mathbb{R}^n$ is cylindrical if:*

1. $n = 1$ and $\mathcal{D}$ is a stack over $\mathbb{R}^0$: a sequence of algebraic points (sections) and open intervals either side of them (sectors),
2. $n > 1$ and there exists a cylindrical decomposition $\mathcal{D}'$ of $\mathbb{R}^{n-1}$ such that $\mathcal{D}$ is a disjoint union of stacks over each cell $C \in \mathcal{D}'$.

In particular, any cylindrical decomposition $\mathcal{D}_n$ of $\mathbb{R}^n$ induces (and is an extension of) the cylindrical decompositions $\mathcal{D}_i$ of $\mathbb{R}^i$ for $i < n$.

Informally, cylindricity means that for any two given cells in $\mathbb{R}^n$, their images under the projection $\pi : \mathbb{R}^n \rightarrow \mathbb{R}^{n-1}$ are identical or disjoint, that is, each higher-dimensional cell is part of a stack over a specific lower-dimensional cell. We say a cell lies *above* a lower-dimensional cell if it is (recursively) contained in such a stack, and *below* a higher-dimensional cell if that cell lies in a stack (recursively) built over it.

Definition 2.5.3 (Semi-algebraic set). *A basic semi-algebraic set $\mathcal{F}_\diamond = \{f_i \diamond_i 0\}$ is a subset of $\mathbb{R}^n$ on which a conjunction of constraints, $\bigwedge (f_i \diamond_i 0)$, is true, for each $f_i \in \mathbb{R}[x_1, \dots, x_n]$. A semi-algebraic set or semi-algebraic system is a union of basic semi-algebraic sets.*

Definition 2.5.4 (Cylindrical Algebraic Decomposition). *A cylindrical algebraic decomposition (CAD) is a cylindrical decomposition whose cells are semi-algebraic sets.*

Definition 2.5.5 (Sign-invariant). Let $S \subseteq \mathbb{R}^n$ and $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$. We say that f is sign-invariant on S if f has constant sign on S , that is:

$$f(\sigma) \diamond 0 \quad \text{for all } \sigma \in S,$$

where $\diamond \in \{<, >, =\}$. In other words, f is either identically zero or never zero on S . The set $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ is sign-invariant on S , and S is $\mathcal{F}$ -(sign-)invariant, if each f_i is sign-invariant on S .

Given a set $\mathcal{F}$ of polynomials in $\mathbb{R}[x_1, \dots, x_n]$ and the associated ordering $x_1 \prec \dots \prec x_n$, a CAD algorithm aims to produce an $\mathcal{F}$ -invariant CAD. We say the CAD of $\mathbb{R}^n$ is $\mathcal{F}$ -invariant if each of its cells is $\mathcal{F}$ -invariant, and will use the notation $\text{CAD}(\mathcal{F})$ to refer to an $\mathcal{F}$ -invariant CAD of $\mathbb{R}^n$.

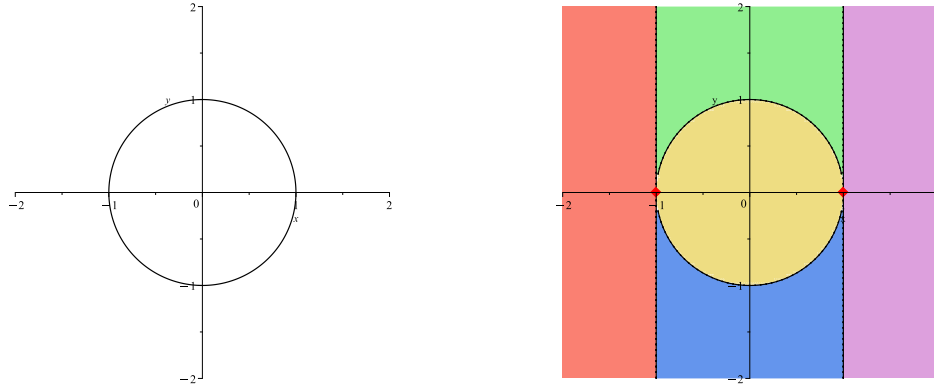


Figure 2-3: The graph of $x^2 + y^2 - 1$ and the $\{x^2 + y^2 - 1\}$ -invariant CAD of $\mathbb{R}^2$.
Figure generated by **GraphCAD** (see Chapter 6).

In one dimension, constructing a sign-invariant CAD is straightforward: its cells are the collection of roots of the polynomials and the regions either side of them. Each sign-invariant cell defines a region where the number of distinct real roots (the ‘root structure’) does not change.

In higher dimensions, we treat the polynomials in $\mathcal{F} = \mathcal{F}_n \subset \mathbb{R}[x_1, \dots, x_n]$ as univariate in the greatest variable with respect to the ordering, written as $\mathcal{F}_n \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$. In order to find when the root structure in x_n changes, we construct a set of *projection polynomials* $\mathcal{F}_{n-1}$ in one fewer variable, whose roots indicate when this change occurs. This process is repeated until we obtain a set of polynomials $\mathcal{F}_1 \subset \mathbb{R}[x_1]$ whose roots and intervals make up a CAD of $\mathbb{R}$.

By the Tarski–Seidenberg Theorem, projections of semi-algebraic sets are again semi-algebraic:

Theorem 2.5.6 (Tarski-Seidenberg [BCR98]). *Let R be a real closed field and $S \subseteq R^n$ a semi-algebraic set. Denote by $\pi : R^n \rightarrow R^{n-1}$ the projection onto the space of the first $n - 1$ coordinates. Then $\pi(S)$ is semi-algebraic.*

Because the projection polynomials ensure the root structure of $\mathcal{F}_k$ does not vary over a cell of the CAD of $\mathbb{R}^{k-1}$, we can continue the process, decomposing the region above the cell into a new set of sign-invariant cells whose union forms the CAD of $\mathbb{R}^n$.

2.5.1 Projection and Lifting

Collins' original CAD algorithm was a *projection and lifting* algorithm which produces a sign-invariant CAD.

This sign-invariance means that one need only check a *sample point* in each cell to determine all possible sign combinations of the input polynomials, as each cell is $\mathcal{F}$ -invariant by construction.

A *sample point* of a cell $C \subseteq \mathbb{R}^k$ is a point $\sigma = (\sigma_1, \dots, \sigma_k) \in C$. The sample points are also cylindrical: the sample points of each cell will be built above those of the cell below.

This projection and lifting algorithm can be split into three phases:

- A **projection phase**, which involves applying a *projection operator* proj to the set of polynomials $\mathcal{F}_n \subset \mathbb{R}[x_1, \dots, x_n]$ to produce a set $\mathcal{F}_{n-1} \subset \mathbb{R}[x_1, \dots, x_{n-1}]$, and repeating this process until one obtains $\mathcal{F}_1 \subset \mathbb{R}[x_1]$. The roots of the polynomials produced by each step are the projections of ‘significant points’ of the previous set of polynomials.
- A **base phase** consisting of real root isolation of $\mathcal{F}_1$, where these roots and the open intervals form an $\mathcal{F}_1$ -invariant CAD of $\mathbb{R}$.
- A **lifting phase** which, for each cell C of the $\mathcal{F}_{k-1}$ -invariant CAD in $\mathbb{R}^{k-1}$, involves constructing a sample point σ and isolating the real roots of the polynomials of $\mathcal{F}_k$ at σ . The sectors and sections of these polynomials form a stack, and these stacks make the cells of the $\mathcal{F}_k$ -invariant CAD of $\mathbb{R}^k$ above C .

Different CAD algorithms of this type may use different projection operators, different invariances, different notions of delineability and different lifting algorithms, depending on their characteristics. Key examples will be explained in detail in Section 2.6.

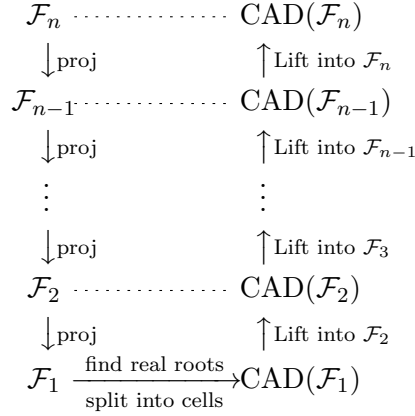


Figure 2-4: Outline of the projection and lifting method for constructing an $\mathcal{F}_n$ -invariant CAD of $\mathbb{R}^n$.

2.5.2 Invariance and Delineability

In a projection and lifting algorithm, cylindricity is ensured through the choice of a specific projection operator proj , which takes a set of polynomials $\mathcal{F}_n \subset \mathbb{R}[x_1, \dots, x_n]$ and produces a set of *projection polynomials* $\mathcal{F}_{n-1} \subset \mathbb{R}[x_1, \dots, x_{n-1}]$ which capture when the root structure of the polynomials in $\mathcal{F}_n$ changes. By repeatedly applying this operator, we produce sets of polynomials in fewer and fewer variables until we obtain a set of polynomials $\mathcal{F}_1 \subset \mathbb{R}[x_1]$. Given this, we can produce a sign-invariant CAD of $\mathbb{R}$ whose cells are the roots of the polynomials of $\mathcal{F}_1$ (where the sign of that polynomial is zero), and the regions either side (where every polynomial is either positive or negative).

Because the polynomials of $\mathcal{F}_1$ are sign-invariant in each cell, the number of roots of the polynomials of $\mathcal{F}_2$ does not change (the polynomials are *delineable*, see Definition 2.5.8). This ensures cylindricity, and guarantees that the polynomials in higher dimensions above each cell (which can be determined by substituting a sample point into the polynomials to obtain a new set of univariate polynomials) will also be sign-invariant. This process can be repeated until we produce our $\mathcal{F}_n$ -invariant CAD of $\mathbb{R}^n$.

As delineability and cylindricity depend on suitable projection polynomials, we must ensure we choose a suitable projection operator. We are interested in choosing projection polynomials which capture when the *root structure* of the polynomials of $\mathcal{F}_n$ changes. Each such polynomial of $\mathcal{F}_n$ would, naturally, be zero on one of its roots (with some multiplicity), and non-zero otherwise. We are also interested when the number of distinct roots in $\mathcal{F}_n$ changes, for example, when two roots coalesce (such as when

two polynomials cross or when a polynomial self-intersects) or when the degree of a polynomial changes.

Definition 2.5.7 (Root structure, root structure matrix). *Let $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ be a polynomial. When taken at a sample point $\sigma \in \mathbb{R}^{n-1}$, $f(\sigma, x_n)$ has degree d and $k \leq d$ distinct real roots $\alpha_1 < \dots < \alpha_k$ with associated positive multiplicities $m_1, \dots, m_k$. The (real) root structure of f over σ consists of the degree, number of distinct real roots and multiplicities of $f(\sigma, x_n)$ and can be represented by the root structure matrix, the $1 \times (k+1)$ matrix $[d \mid m_1 \cdots m_k]$, which is $[0]$ if $f(\sigma, x_n)$ is a non-zero constant, or $[-\infty]$ if it is identically zero.*

For a set of polynomials $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, we consider the collection of roots among the f_i , or equivalently the roots $\alpha_1 < \dots < \alpha_k$ of $\prod f_i$. The root structure of $\mathcal{F}$ over σ consists of the degree, number of distinct real roots and multiplicities of each $f_i(\sigma, x_n)$, and the common root multiplicity between them. The root structure matrix of $\mathcal{F}$ over σ is the $r \times (k+1)$ matrix in which row i is the root structure matrix of f_i over σ , column 1 is the degree of each $f_i(\sigma, x_n)$ and column $j+1$, for $1 \leq j \leq k$, contains the multiplicities of the root α_j in each f_i (which may be zero):

$$\left[\begin{array}{c|ccc} d_1 & m_{1,1} & \dots & m_{1,k} \\ d_2 & m_{2,1} & \dots & m_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ d_r & m_{r,1} & \dots & m_{r,k} \end{array} \right]$$

Using this allows us to store and retrieve key information:

- The number of real roots of f_i over σ (with multiplicity) is $\sum_j m_{i,j}$.
- The number of distinct real roots of f_i over σ is the number of non-zero $m_{i,j}$.
- The number of common real roots (with multiplicity) between f_p and f_q over σ ($p < q$) is $\sum_j \min(m_{p,j}, m_{q,j})$.

Further, the root structure matrix of $\prod f_i$ over σ , the $1 \times (k+1)$ matrix $[d \mid m_1 \cdots m_k]$, can be obtained from the root structure matrix of $\mathcal{F}$ over σ by taking the sum of the entries in each column.

By choosing suitable projection polynomials, we can produce cells in which the root structure does not change, ensuring that each polynomial in $\mathcal{F}_n$ will have a fixed number of roots on any point of these regions.

In order to construct a satisfactory CAD, we need a projection that ensures *delineability*.

Definition 2.5.8 (Delineable [Jir95]). *Let $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ be a set of real polynomials, and let $S \subseteq \mathbb{R}^{n-1}$. We say that $\mathcal{F}$ is delineable over S if the following conditions hold for all $\sigma \in S$:*

1. *The total number of complex roots of $f_i(\sigma, x_n)$ (counting multiplicity) remains constant.*
2. *The number of distinct complex roots of $f_i(\sigma, x_n)$ remains constant.*
3. *For each $i < j$, the total number of common complex roots of $f_i(\sigma, x_n)$ and $f_j(\sigma, x_n)$ (counting multiplicity) remains constant.*

However, we are interested specifically in real roots, and present an alternate definition, as given by Collins:

Definition 2.5.9 (Delineable [Col98]). *Let $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, $n \geq 2$, and let $S \subseteq \mathbb{R}^{n-1}$. We say that f is delineable on S , and the real-valued functions $\alpha_1, \dots, \alpha_k$, $k \geq 0$, delineate the real roots of f on S if the following all hold for every $\sigma \in S$:*

1. *There exist $\ell \geq 0$ positive integers $m_1, \dots, m_\ell$ such that the univariate polynomial $f(\sigma, x_n)$ has exactly ℓ distinct complex roots, with multiplicities $m_1, \dots, m_\ell$.*
2. *For $k \leq \ell$, the functions $\alpha_1 < \alpha_2 < \dots < \alpha_k$ are continuous.*
3. *$\alpha_i(\sigma)$ is a real root of $f(\sigma, x_n)$ of multiplicity m_i for $1 \leq i \leq k$.*
4. *If $\sigma \in S, b \in \mathbb{R}$ and $f(\sigma, b) = 0$ then $b = \alpha_i(\sigma)$ for some $1 \leq i \leq k$.*

In short, the number of distinct real roots and their individual multiplicities do not change over S . Recall that, in fact, these functions are the same functions from Definition 2.5.1.

If $\mathcal{F} = \{f_1, \dots, f_r\}$ is a non-empty finite set of non-zero polynomials, we will say that $\mathcal{F}$ is delineable over $S \subseteq \mathbb{R}^{n-1}$ when $\prod_{i=1}^n f_i$ is delineable over S . Note that each f_i could be delineable over S without $\mathcal{F}$ being delineable over S .

Note that Definition 2.5.8 gives a computable, sufficient definition for delineability, but also one that can be violated at non-real points. On the other hand, Definition 2.5.9 gives a definition in terms of real roots, but is untestable. However, we may use the former to ensure real root structure is fixed.

Theorem 2.5.10 (Delineability and Real Root Structure). *Let $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ and let S be a connected subset of $\mathbb{R}^{n-1}$. If f is delineable over S as per Definition 2.5.8, then the real root structure of f over S is constant (that is, the real root structure of f over σ is the same for every $\sigma \in S$), and f is delineable over S as per Definition 2.5.9.*

Proof. [Col98, p.89], or the following. Suppose that $f(\sigma, x_n)$ is a degree d polynomial in x_n with $\text{lc}(f(\sigma, x_n)) \neq 0$ for all $\sigma \in S$. Then $f(\sigma, x_n)$ has d complex solutions for any $\sigma \in S$. Suppose also that the number of distinct complex roots is independent of σ and is equal to ℓ .

We shall first show that delineability of f over S ensures the *complex* root structure of f over S does not change. Recall that if a set S is connected, all continuous functions from S to discrete space are constant. We claim that complex root structure varies continuously with σ .

As with real root structure, the complex root structure of f over S is a finite list of integers representing the degree of f and the multiplicities of each complex root, ordered in some way. Unlike real roots, complex roots have no specified global ordering. However, continuity is a local property, so we only need to consider this for a single fixed point in S , which we shall take to be $\mathbf{0}$.

If we assume $f(\mathbf{0}, x_n)$ has the distinct complex roots $\alpha_1, \dots, \alpha_\ell$, we may construct some sufficiently general real line in $\mathbb{C}$ passing through the origin, such that the projections of $\alpha_1, \dots, \alpha_\ell$ to this line are distinct. We may then order the roots according to their position on this line, ensuring locally consistent labelling.

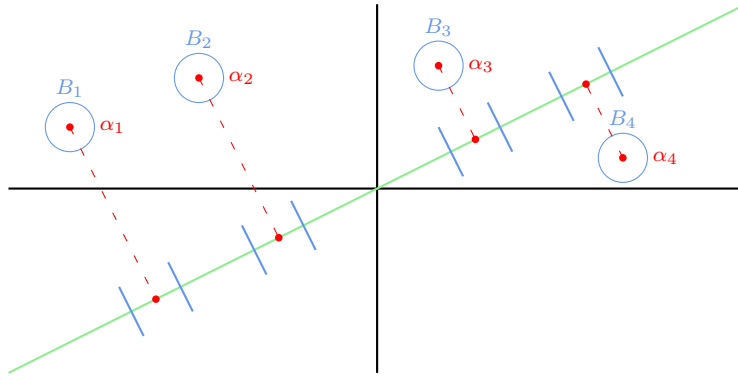


Figure 2-5: The roots of $f(\mathbf{0}, x_n)$, ordered according to their positions on a general real line through the origin.

We now show that the solutions to $f(\sigma, x_n)$ vary continuously with σ and root multiplicities do not change. Take an ϵ -ball B_i around each α_i , sufficiently small enough to ensure that the balls, and their images on the line, are disjoint. Suppose that each α_i

has multiplicity m_i , then by Rouché's theorem, $f(\sigma, x_n)$ also has roots inside B_i whose total multiplicity is also m_i , for $\|\sigma\| \ll 1$.

There must be at least one distinct root of $f(\sigma, 0)$ inside B_i (as $m_i > 0$), and $f(\sigma, 0)$ must have ℓ distinct roots in total. Therefore, each B_i must contain exactly one distinct root of $f(\sigma, x_n)$ with multiplicity m_i .

We now consider real root structure and assume that f has real coefficients and that σ and x_n are real. For $\|\sigma\| \ll 1$, if α_i is a real root of $f(0, x_n)$ of multiplicity m_i , then by the above, there is a ball $B_i (\subset \mathbb{C})$ around α_i in which $f(\sigma, x_n)$ has exactly one root. If this root were non-real, its conjugate would also lie in B_i , increasing the number of distinct roots in the ball and in total. Thus, this root must be real, and the number of distinct real roots, and their individual multiplicities, are constant over S . $\square$

Remark 2.5.11. *Note that the number of distinct complex roots must remain constant to fix the real root structure; it is not sufficient to have a constant number of distinct real roots. Consider the example $f(x, y) = x^3 + y$. This has a single distinct real root for all $y \in \mathbb{R}$, but this is a triple root at $y = 0$ and a simple root elsewhere.*

We aim to construct cells C such that the polynomials in $\mathcal{F}$ are delineable over each C . To do so, we must identify where the root structure of these polynomials may change:

1. The total number of complex roots of $f_i(\sigma, x_n)$ (counted with multiplicity) is equal to $\deg(f_i)$, and this can only drop when the degree drops.
2. The number of distinct complex roots of $f_i(\sigma, x_n)$ can change when roots merge or split, so we must look at when f_i has repeated roots.
3. The number of common complex roots between $f_i(\sigma, x_n)$ and $f_j(\sigma, x_n)$ (counting multiplicity) changes when the polynomials share a root.

In short, we are looking to find values for x_n (as polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}]$) which capture when the root structure of the polynomials in $\mathcal{F}$ changes. The first of the above cases is captured by each leading coefficient, as the degree of a polynomial f can only drop when $\text{lc}(f) = 0$. The remaining two cases, when two polynomials cross or when a polynomial has multiple roots, are captured by *resultants* and *discriminants* respectively. Like the leading coefficient, these are polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}]$ and are zero precisely at these points.

Different projection operators inform different types of invariance, and in turn their own lifting algorithms and notions of delineability. Given an appropriate projection operator proj and a type of invariance (see Section 2.6 for examples of these), we can produce a CAD if the following holds:

Theorem 2.5.12 (Main CAD Theorem). *Let $\mathcal{F}$ be a non-empty set of non-zero real polynomials in n real variables, $n \geq 2$. Let S be a connected subset of $\mathbb{R}^{n-1}$. Let $\text{proj}(\mathcal{F})$ be the projection of $\mathcal{F}$. If every element of $\text{proj}(\mathcal{F})$ is sign-invariant on S , then the polynomials in $\mathcal{F}$ are delineable over S .*

2.5.3 The Sylvester Matrix

We now look at the cases where the number of distinct real roots may change, for which we need the following definitions.

Definition 2.5.13 (Sylvester Matrix). *Suppose $f, g \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ have degree d and e respectively. The Sylvester matrix of f and g , $\text{Syl}(f, g)$ is the $(d+e) \times (d+e)$ matrix whose rows contain the coefficients of the polynomials $x_n^{e-1}f, \dots, f, x_n^{d-1}g, \dots, g$, with the coefficients of x_n^i occurring in column $d+e-i$.*

If $f = a_d x_n^d + a_{d-1} x_n^{d-1} + \dots + a_1 x_n + a_0$ and $g = b_e x_n^e + b_{e-1} x_n^{e-1} + \dots + b_1 x_n + b_0$, then

$$\text{Syl}(f, g) = \underbrace{\begin{pmatrix} a_d & a_{d-1} & \cdots & a_0 & 0 & \cdots & \cdots & 0 \\ 0 & a_d & a_{d-1} & \cdots & a_0 & 0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & & \ddots & & \vdots \\ 0 & \cdots & \cdots & 0 & a_d & a_{d-1} & \cdots & a_0 \\ b_e & b_{e-1} & \cdots & b_0 & 0 & \cdots & \cdots & 0 \\ 0 & b_e & b_{e-1} & \cdots & b_0 & 0 & \cdots & 0 \\ \vdots & \ddots & & \ddots & & \ddots & & \vdots \\ 0 & \cdots & \cdots & 0 & b_e & b_{e-1} & \cdots & b_0 \end{pmatrix}}_{d+e} \left. \begin{matrix} \left. \begin{matrix} \vdots \\ \vdots \\ \vdots \end{matrix} \right\} e \\ \left. \begin{matrix} \vdots \\ \vdots \\ \vdots \end{matrix} \right\} d \end{matrix} \right\}$$

We allow $d = 0$ or $e = 0$ and adopt the convention $\det(N) = 0$ if N is a 0×0 matrix.

Definition 2.5.14 (Resultant, discriminant). *For $f, g \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, we define the resultant of f and g , denoted $\text{res}(f, g)$, as the determinant of $\text{Syl}(f, g)$. The discriminant of f , denoted $\text{disc}(f)$, is $\text{res}(f, f')$, where f' is the derivative of f with respect to x_n . For $\mathcal{F}, \mathcal{G} \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, we define*

- $\text{res}(\mathcal{F}, \mathcal{G}) = \{\text{res}(f_i, g_j) \mid f_i \in \mathcal{F}, g_j \in \mathcal{G}, f_i \neq g_j\},$
- $\text{res}(\mathcal{F}) = \text{res}(\mathcal{F}, \mathcal{F}) = \{\text{res}(f_i, f_j) \mid f_i, f_j \in \mathcal{F}, i < j\},$
- $\text{disc}(\mathcal{F}) = \{\text{disc}(f) \mid f \in \mathcal{F}\}.$

In cases when a variable needs to be specified for these operators, it will be denoted in subscript.

Definition 2.5.15 (Subresultant, PSC). For $0 \leq i \leq j \leq \min(d, e)$ let $M_{j,i}$ be the matrix obtained from $\text{Syl}(f, g)$ by deleting the last j rows of f coefficients, the last j rows of g coefficients, and all of the last $2j + 1$ columns except column $d + e - i - j$.

The j th subresultant of f and g is the polynomial $S_j(f, g) = \sum_{i=0}^j \det(M_{j,i}) x_n^i$, which has degree at most j and the j th principal subresultant coefficient (PSC) of f and g , $\text{psc}_j(f, g)$, is the coefficient of x_n^j in $S_j(f, g)$, i.e. $\text{psc}_j(f, g) = \det(M_{j,j})$.

Note that $M_{0,0} = \text{Syl}(f, g)$ and $S_0(f, g) = \text{res}(f, g)$.

Theorem 2.5.16 (Subresultants and GCD). Let f and g be non-zero polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}][x_n]$. Then $\deg(\gcd(f, g)) = k$ if and only if $\text{psc}_k(f, g)$ is the first non-zero PSC, that is, k is the smallest number such that $\text{psc}_k(f, g) \neq 0$ and $\text{psc}_j(f, g) = 0$ for all $j < k$. Moreover, $S_k(f, g)$ is a scalar (in $\mathbb{R}[x_1, \dots, x_{n-1}]$) multiple of $\gcd(f, g)$.

Theorem 2.5.17 (GCD and Distinct Roots). Let $f(x)$ be a real univariate polynomial with $\deg(f) = d \geq 1$ and $k = \deg(\gcd(f, f'))$. Then $d - k$ is the number of distinct (complex) roots of f .

Indeed, if a polynomial has repeated factors, these factors will also appear in its derivative. If f and its derivative are coprime, each complex root (and hence each real root) appears exactly once. From this we have the following result:

Corollary 2.5.18 (Resultants, Discriminants, and Roots). Let f and g be non-zero polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}][x_n]$. The resultant $\text{res}(f, g)$ is zero whenever f and g have a common root, and the discriminant $\text{disc}(f)$ is zero whenever f has a repeated root.

2.5.4 Basis, Primitive Part, Content, Squarefree

As a consequence of Corollary 2.5.18, if the polynomials in $\mathcal{F}$ have common factors or repeated roots, the associated resultants and discriminants would be identically zero. In order to avoid difficulties caused by this, we make some standard simplifications: inputs are usually assumed to be squarefree, coprime and primitive. If they are not, there are algorithmic ways to produce an equivalent input that is.

Definition 2.5.19 (Content, primitive part). We define the following (with respect to x_n) for each non-zero polynomial $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$:

- The content of f , $\text{cont}(f) = \gcd(\text{coeffs}(f))$.
- The primitive part of f , $\text{pp}(f) = f / \text{cont}(f)$.

Similarly, for a collection of polynomials $\mathcal{F} \subset \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, we define

- $\text{cont}(\mathcal{F}) = \{\text{cont}(f) \mid f \in \mathcal{F}\},$
- $\text{pp}(\mathcal{F}) = \{\text{pp}(f) \mid f \in \mathcal{F}\},$

We say f is *primitive* if $\text{cont}(f) = 1$. Recall Gauss' Lemma: the product of two primitive polynomials is primitive.

Remark 2.5.20. If f and g are primitive polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, then Gauss' Lemma implies that $\gcd(f, g) = S_k(f, g)$. In general,

$$\gcd(f, g) = \gcd(\text{cont}(f), \text{cont}(g)) \cdot \gcd(\text{pp}(f), \text{pp}(g)).$$

Similarly, if f and g have degrees d and e respectively:

$$\text{res}(f, g) = \text{res}(\text{cont}(f) \text{pp}(f), \text{cont}(g) \text{pp}(g)) = \text{cont}(f)^d \text{cont}(g)^e \text{res}(\text{pp}(f), \text{pp}(g)),$$

$$\text{disc}(f) = \text{disc}(\text{cont}(f) \text{pp}(f)) = \text{cont}(f)^{2d-2} \text{disc}(\text{pp}(f)).$$

Definition 2.5.21 (Basis). In the following we take A to be a unique factorisation domain such as $\mathbb{R}$ or a polynomial ring over $\mathbb{R}$. If $\mathcal{F}$ is a set of non-constant primitive polynomials of positive degree over A , then a basis for $\mathcal{F}$ is a set $\mathcal{B}$ of primitive coprime polynomials of positive degree over A such that every element of $\mathcal{F}$ is a unit multiple of some product of elements of $\mathcal{B}$. If the elements of $\mathcal{F}$ are not primitive, a basis for $\mathcal{F}$ is the union of $\text{cont}(\mathcal{F})$ and elements of a basis of $\text{pp}(\mathcal{F})$.

Among all bases of a set $\mathcal{F}$, there is a *finest basis* $\mathcal{B}_F$ and a *coarsest basis* $\mathcal{B}_C$, whose elements are respectively the irreducible divisors of, and the products of, the elements of every other basis.

The finest basis $\mathcal{B}_F$ is the basis consisting of the irreducible factors of elements of $\mathcal{F}$. The coarsest basis $\mathcal{B}_C$ is the minimal set of coprime polynomials whose product structure recovers all elements of $\mathcal{F}$. An *irreducible basis* of $\mathcal{F}$ is one whose elements are irreducible, and naturally $\mathcal{B}_F$ is an irreducible basis. The elements of $\mathcal{B}_F$ and $\mathcal{B}_C$ are unique up to multiplication of units of A .

If $\mathcal{F}$ is finite, $\mathcal{B}_C$ can be obtained by repeated GCD separation: for $f, g \in \mathcal{F}$ with $G = \gcd(f, g) \neq 1$, replace f and g by the non-constant polynomials among G , f/G , and g/G until only coprime elements remain, giving $\mathcal{B}_C$.

Definition 2.5.22 (Squarefree). *A polynomial is squarefree if each of its irreducible factors occurs with multiplicity one. A squarefree basis of $\mathcal{F}$ is a basis whose elements are squarefree.*

Every primitive polynomial f admits a *squarefree factorisation*

$$f = c \prod_{i=1}^k s_i^{m_i}$$

where $s_1, \dots, s_k$ are squarefree, coprime polynomials, $m_1, \dots, m_k$ their multiplicities, and $c \in \mathbb{R}$. The set $s = \{s_1, \dots, s_k\}$ is the coarsest squarefree basis for $\{f\}$.

One way to obtain s is to find a squarefree polynomial for which s is also the coarsest squarefree basis. Recall that any polynomial with repeated factors shares those factors with its derivative. Hence, from f one can produce the squarefree polynomial $\hat{f} = f / \gcd(f, f')$ which has the same squarefree basis.

One can find the coarsest squarefree basis for $\mathcal{F}$ by finding the coarsest squarefree basis for each $f \in \mathcal{F}$ and then applying GCD separation to their union.

The coarsest squarefree basis of $\mathcal{F}$ is quicker to compute, but may contain many factors. In contrast, its finest squarefree basis is the irreducible basis $\mathcal{B}_F$, and while this may take longer to compute, it is often easier to work with. In terms of implementation, the irreducible basis for $\mathcal{F}$ is the union of the irreducible bases of each f , and is easier to update when presented with new polynomials.

The motivation for ensuring our polynomials are primitive, squarefree and coprime is summed up nicely by the following consequence of Corollary 2.5.18.

Corollary 2.5.23 (Resultants, Discriminants, Coprime, and Squarefree). *Let f and g be primitive non-zero polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}][x_n]$. Then*

- *f and g are coprime if and only if $\text{res}(f, g) \neq 0$, and*
- *f is squarefree if and only if $\text{disc}(f) \neq 0$.*

The projection operators and their associated theorems and algorithms assume that the input polynomials are squarefree and coprime. If they are not, the polynomials can be made so, such as by taking an irreducible basis. Additionally, these polynomials are made primitive: any $\text{cont}(f) \neq 0$ appears as a factor of every projection polynomial, and since we are interested in the roots of the projection polynomials, it suffices to compute $\text{cont}(f) \cup \text{proj}(\text{pp}(f))$.

2.6 Projection and Lifting CAD Algorithms

As previously discussed, projection polynomials must capture the points at which the root structure changes, such as degree drops and root crossings. The first viable algorithm for CAD, due to Collins [Col75], uses a large projection operator consisting of the coefficients and PSCs of input polynomials and their *reducta* (polynomials obtained by repeatedly removing leading terms) to ensure a sign-invariant CAD. McCallum [McC84] later introduced a reduced projection operator composed of the coefficients, discriminants, and resultants of the input polynomials, relying instead on *order-invariance*. However, construction of an order-invariant CAD fails in the presence of *curtains* [NDS20] (fibres of projections on which a relevant polynomial is identically zero). Lazard [Laz94], later validated in [MPP19], proposed a further refined projection operator requiring *lex-least-invariance* and the associated notion of *Lazard delineability*, which is defined even in the presence of curtains. Further refinements, such as [BM20], are also in use.

For a more detailed description of these approaches, operators and invariances, see [Col75, McC84, Laz94, McC99, McC01, MPP19, Nai21, DNSU23].

2.6.1 Projection Operators, Lifting and Invariance

Definition 2.6.1 (General projection operator). *In general, for a set of polynomials $\mathcal{F} \subset \mathbb{R}[x_1, \dots, x_n]$, the projection operator $\text{proj}(\mathcal{F}) \subset \mathbb{R}[x_1, \dots, x_{n-1}]$ has the form*

$$\text{proj}(\mathcal{F}) := \text{res}(\mathcal{F}) \cup \text{disc}(\mathcal{F}) \cup \text{projterms}(\mathcal{F}),$$

where:

- for the Collins projection $\text{PC}(\mathcal{F})$, we take

$$\text{projterms}(\mathcal{F}) = \text{coeffs}(\mathcal{F}) \cup \{\text{psc}_k(b_i, b'_i)\} \cup \{\text{psc}_k(b_i, b_j)\},$$

for $k \geq 0$ and b_i, b_j either polynomials in $\mathcal{F}$ or their reducta (barring the case where $k = 0$ and $b_i, b_j \in \mathcal{F}$),

- for the McCallum projection $\text{PM}(\mathcal{F})$, we take $\text{projterms}(\mathcal{F}) = \text{coeffs}(\mathcal{F})$,
- for the Lazard projection $\text{PL}(\mathcal{F})$, we take $\text{projterms}(\mathcal{F}) = \text{lc}(\mathcal{F}) \cup \text{tc}(\mathcal{F})$.

Each of these projection operators produces polynomials which are invariant with respect to some property (sign, order and lex-least valuation). These properties are *valuations*:

Definition 2.6.2 (Valuation). *A valuation is a map ν from $\mathbb{R}[x_1, \dots, x_n]$ to G a totally ordered additive semigroup that satisfies the following:*

- $\nu(f) = \infty$ if and only if $f = 0$,
- $\nu(fg) = \nu(f) + \nu(g)$,
- $\nu(f + g) \geq \min(\nu(f), \nu(g))$, with equality if $\nu(f) \neq \nu(g)$.

We use $\nu_\sigma(f)$ to denote the valuation of $f(\sigma)$, for $\sigma \in \mathbb{R}[x_1, \dots, x_n]$.

Definition 2.6.3 (Valuation-invariant). *Let $S \subseteq \mathbb{R}^n$ and $f \in \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$. We say that f is valuation-invariant or ν -invariant on S if the valuation of f is constant throughout S . The set $\mathcal{F} = \{f_1, \dots, f_r\}$ of polynomials in $\mathbb{R}[x_1, \dots, x_{n-1}][x_n]$ is valuation-invariant on S (and S is $\mathcal{F}$ -valuation-invariant) if each f_i is valuation-invariant on S .*

These projections and valuations have an associated notion of invariance and delineability, under which a general lifting theorem holds:

Theorem 2.6.4 (General Lifting [Col75, McC99, MPP19]). *Let $\mathcal{F} = \{f_1, \dots, f_r\}$ be a set of pairwise coprime irreducible polynomials in $\mathbb{R}[x_1, \dots, x_n]$ of positive degree in x_n , for $n \geq 2$. Let S be a connected subset of $\mathbb{R}^{n-1}$ on which each element of $\text{proj}(\mathcal{F})$ is ν -invariant. Then every element of $\mathcal{F}$ that is not identically zero on S is ν -delineable over S , and the ν -sections over S are pairwise disjoint.*

- For the *Collins* projection operator, this theorem uses sign-invariance, and the established notions of delineability and sections.
- For the *McCallum* projection operator, this theorem requires the stronger notion of order-invariance and analytic delineability.
- For the *Lazard* projection operator, this theorem requires *lex-least-invariance*, and the notions of *Lazard delineability* and *Lazard sections*.

The importance of elements being *not identically zero* is vastly different between these three projection operators and notions of invariance: for sign-invariance, Collins [Col75] notes that if a polynomial f is identically zero on S , it is not delineable but any decomposition of the cylinder of S will naturally be $\{f\}$ -invariant, so f can be effectively ignored. In the case of order-invariance [McC85], if a polynomial nullifies in S then order-invariance is not guaranteed over S , so in fact this *only* holds if none of the polynomials nullify on S (a condition known as *well-orientedness*). For the Lazard projection and lex-least-invariance [MPP19, Nai21], Lazard delineability holds even for polynomials which are identically zero on S , so this theorem is true for every element.

These projection operators can be further simplified to produce a smaller CAD in which the original QE formula is true in the same places. Methods for doing this are discussed in Section 2.8. Note that both order- and lex-least-invariance imply sign-invariance, allowing these improved projections to be used for QE.

2.7 Variable Ordering Heuristics

The choice of variable ordering can have substantial impact on the cell count of a CAD, and consequently on algorithmic performance, with a well-chosen variable ordering yielding significant improvements. In some situations the ordering is partially or entirely fixed (see Section 2.8.3), but when freedom exists a method is needed to select a sensible order. Many features can affect the complexity of a CAD, most notably polynomial degree, which is directly tied to the number of possible roots. Since projection can rapidly increase both the maximum degree and the number of polynomials, a natural idea is to project the variable with the lowest degree first to keep degree growth down. This has motivated the creation of several heuristics.

These heuristics look at features of the input polynomials, but not the overall structure of the input formula. Ideas for how to consider this structure are discussed in Chapter 3.

Brown [Bro04] gives the following heuristic:

Definition 2.7.1 (Brown heuristic). *For $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_n]$ a set of polynomials, the Brown heuristic suggests choosing variables according to the following rules, applied in order:*

- *lowest degree in $\mathcal{F}$,*
- *lowest total degree among the terms in which the variable appears,*
- *lowest number of terms containing the variable.*

Despite its relative simplicity, the Brown heuristic has been shown to perform well in practice [HEW⁺19].

Another notable heuristic is **sotd** (sum of the degrees) [DSS04].

Definition 2.7.2 (sotd heuristic). *Let $\mathcal{P} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$ be the sets of projection polynomials generated under the ordering $x_1 \prec \dots \prec x_n$, and let each $\mathcal{F}_i$ be a set of polynomials in the variables $x_1, \dots, x_i$. The function **sotd** takes the sum of all total degrees of all monomials of all polynomials in all possible $\mathcal{P}$:*

$$\text{sotd}(\mathcal{P}) = \sum_{f \in \mathcal{P}} \sigma(f),$$

where $\sigma(f)$ is the sum of the total degrees of all monomials in f . The **sotd** heuristic takes the ordering with the smallest $\text{sotd}(\mathcal{P})$.

As this requires constructing $\mathcal{P}$ for each of the $n!$ possible orderings, the method is practical only for small n . The *greedy* version of **sotd**, **gsotd** (referred to in the literature as **greedy**) instead performs **sotd** for each step instead of for each full $\mathcal{P}$: by first projecting with respect to each variable, keeping the projection with the lowest $\sum \sigma(f)$, and repeating this process on the projection polynomials.

The **mods** and **gmods** heuristics, introduced in [dRE22], are obtained as follows. For a set of polynomials $\mathcal{F}_k = \{f_{k,1}, \dots, f_{k,r_k}\} \in \mathbb{R}[x_1, \dots, x_k]$, define the *degree sum* of x_k as

$$D_{x_k}(\mathcal{F}_k) = \sum_{i=1}^{r_k} \deg_{x_k}(f_{k,i}).$$

Clearly, $D_{x_k}(\mathcal{F}_k)$ is the maximum number of distinct roots in $\mathcal{F}_k$. Lifting a CAD of $\mathcal{F}_1$ would produce at most $2D_{x_1}(\mathcal{F}_1)+1$ cells, and lifting a CAD of $\mathcal{F}_2$ would involve creating at most $2D_{x_2}(\mathcal{F}_2)+1$ cells over each cell of the CAD of $\mathcal{F}_1$. Therefore, the upper bound for the total number of cells produced at the end of the lifting stage is

$$\prod_{i=1}^n (2D_{x_i}(\mathcal{F}_i) + 1).$$

As the number of cells in a CAD is strongly correlated with the time needed to build it, this motivates choosing an ordering that minimises the maximum number of cells in the final CAD.

Definition 2.7.3 (**mods** heuristic). For $\mathcal{P} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$ the sets of projection polynomials generated under a variable ordering, and each $\mathcal{F}_i$ a set of polynomials in i variables, we define the *multiplication of degree sum function* as:

$$\text{mods}(\mathcal{P}) := \prod_{i=1}^n D_{x_i}(\mathcal{F}_i) = \prod_{i=1}^n \sum_{j=1}^{r_i} \deg_{x_i}(f_{i,j}).$$

The **mods** heuristic chooses the variable order that results in the lowest value of $\text{mods}(\mathcal{P})$.

As with **sotd**, the full **mods** heuristic requires testing all $n!$ orderings, making this heuristic feasible only for low values of n . The greedy variant, **gmods**, instead determines the variable ordering by recursively projecting with respect to the variable with the lowest degree sum. As this requires only the projections needed to build a single CAD, it is considerably quicker. This heuristic is also quicker than **gsotd**, relying on a property which is quick and easy to compute and minimising the number of projections performed. This is the heuristic chosen for the implementation discussed in Chapter 4.

2.8 Formula-specific Improvements

So far, we have viewed CAD primarily as a decomposition with respect to a *set of polynomials*. However, in quantifier elimination the input is a *logical formula* built from polynomial constraints, motivating improvements that take into account the structure of the Tarski formula Φ , rather than just its constituent polynomials. The central concept is *truth-invariance*: instead of requiring sign-, order-, or lex-least-invariance for each polynomial, we require only that the truth value of the formula be invariant within each cell.

Many improvements can be made with truth-invariance in mind. For example, there is no need to further decompose regions in which the formula is known to be false, and computation may be able to terminate early if a region in which the formula is true is found. One powerful technique involves exploiting equational constraints (ECs): given a conjunction of constraints that includes an equation $f = 0$, the conjunction is false in regions where $f \neq 0$, and these regions need no further decomposition.

A refinement of this is *truth-table-invariance*. When a formula is expressed in disjunctive normal form, we may consider the truth values for each conjunction separately, and can make further simplifications.

2.8.1 Truth-Invariance

When solving a QE problem via CAD, the end goal is not the invariance of the polynomials, but the invariance of the *truth value* of the formula. The CAD construction methods discussed so far take as input a collection of polynomials $\mathcal{F}$ obtained from a Tarski formula Φ , without considering the constraints on the polynomials or how they appeared in Φ .

The remainder of this section discusses how information from Φ can be exploited to simplify the CAD process and reduce redundant calculation, taking semi-algebraic systems as input and producing a CAD that is *truth-invariant* with respect to Φ .

Definition 2.8.1 (Truth-invariance). *Let $S \subseteq \mathbb{R}^n$ and $f \in \mathbb{R}[x_1, \dots, x_n]$. We say that the polynomial constraint $f \diamond 0$ is truth-invariant in S if $f \diamond 0$ has constant truth value (identically true or identically false) throughout S . For a set $\mathcal{F}_\diamond = \{f_i \diamond_i 0\}$, we say that $\mathcal{F}_\diamond$ is truth-invariant in S , and S is $(\mathcal{F}_\diamond\text{-})$ truth-invariant, if every constraint of $\mathcal{F}_\diamond$ is truth-invariant in S . A decomposition is $(\mathcal{F}_\diamond\text{-})$ truth-invariant if each of its cells is truth-invariant.*

Remark 2.8.2. *It is worth noting that truth-invariance is defined for polynomial constraints, whereas the previous notions of invariance discussed in Section 2.6.1 apply to polynomials. Polynomial (sign-)invariance implies truth-invariance for its constraints: for $\mathcal{F}$ and S as given above, if $\mathcal{F}$ is valuation-invariant on S , then any $\mathcal{F}_\diamond = \{f_i \diamond_i 0\}$, and in fact any unquantified formula Φ of constraints on elements of $\mathcal{F}$, is truth-invariant on S .*

Definition 2.8.3 (Solution set). *Let $f \diamond 0$ be a polynomial constraint for the polynomial $f \in \mathbb{R}[x_1, \dots, x_n]$. The solution set $Z(f \diamond 0)$ is the set of points in which $f \diamond 0$ is true. For ϕ a quantifier-free Tarski formula of polynomial constraints, $Z(\phi)$ is the set of points in which ϕ is true.*

In particular, we refer to $Z(f = 0)$ as the zero set of f , denoted V_f . For a set of polynomials $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x_1, \dots, x_n]$, the zero set $V_{\mathcal{F}} = Z(\bigwedge (f_i = 0)) = \bigcap Z(f_i = 0)$ is the set of common zeroes of each f_i .

Recall from Definition 2.5.3 that the basic semi-algebraic set $\mathcal{F}_\diamond = \{f_i \diamond_i 0\}$ is the solution set $Z(\varphi)$ for its associated formula $\varphi = \bigwedge (f_i \diamond_i 0)$. Similarly, any DNF quantifier-free Tarski formula will have a semi-algebraic set as its solution set, that is, for ϕ in the form

$$\phi = \bigvee_i \bigwedge_j (f_{i,j} \diamond_{i,j} 0),$$

we have

$$Z(\phi) = \bigcup_i \bigcap_j Z(f_{i,j} \diamond_{i,j} 0).$$

This also highlights how an appropriate CAD can be used to solve any QE problem involving such a ϕ : for $f \in \mathbb{R}[x_1, \dots, x_n]$, the polynomial f can be expressed as $(f > 0) \vee (f = 0) \vee (f < 0)$ and

$$Z(f) = (Z(f > 0) \cup Z(f = 0) \cup Z(f < 0)) = \mathbb{R}^n.$$

In the same way, a set $\mathcal{F}$ of polynomials without constraints is the disjunction of all possible combinations of polynomial sign constraints, and its solution set $\mathbb{R}^n$ can be decomposed into the semi-algebraic sets associated with each combination.

A CAD is one such decomposition of $\mathbb{R}^n$ with the additional property of cylindricity, and by constructing the CAD of $\mathbb{R}^n$ which is invariant with respect to a set of polynomials, one also produces as a subset the semi-algebraic system for any quantifier-free Tarski formula involving those polynomials.

Note, however, there are additional considerations when dealing with quantifiers. As the aim of QE is to take a quantified Tarski formula and produce a logically equivalent formula in the free variables only, the problem formulation naturally imposes a partial projection order for CAD construction. For a prenex Tarski formula

$$\Phi = Q_{k+1}x_{k+1} \dots Q_n x_n \phi(x_1, \dots, x_n), \quad (2.4)$$

quantified variables must be ordered above free variables, and in the order dictated by the prefix. Constructing a CAD of $\mathbb{R}^n$ that respects this order ensures that one can assemble the quantifier-free equivalent formula Ψ using constraints on polynomials in $\mathcal{F}_k$, informed by the true cells of $\text{CAD}(\mathcal{F}_k)$.

As in the quantifier-free case, we must first determine the truth values for the cells of $\text{CAD}(\mathcal{F}_n)$, and from there, the truth values are propagated down, informed by the quantifier Q_n : if $Q_n = \exists$ and the current cell is true, then the cell below it is also true, and if $Q_n = \forall$ and the current cell is false, the cell below it is false. Otherwise, the truth of a cell of $\text{CAD}(\mathcal{F}_{n-1})$ is determined by the truth values for every cell in the stack above it: if $Q_n = \forall$ and every cell above it is true, the cell is true, otherwise it is false. Similarly, if $Q_n = \exists$ and every cell above is false, the cell is false. This process is repeated using every remaining quantifier Q_{n-1} to Q_{k+1} until the truth values for the cells of $\text{CAD}(\mathcal{F}_k)$ are determined, and the combinations of constraints on the polynomials in $\mathcal{F}_k$ in the cells which are true make up Ψ .

2.8.2 Partial CAD

An early attempt to exploit the structure of the formula, and to answer the question about the truth value of the formula, was *Partial CAD* [Hon90, CH91]. This uses the same projection and lifting structure, with no change to the projection phase, but a more intelligent lifting.

- For an existential quantifier $\exists$, we can stop considering the sectors/sections in a cylinder as soon as one is found to be true (and for $\forall$ we can stop when one is found to be false).
- If x is free, we have $\forall y f(x) \wedge g(x, y)$ and on an x -sector (or section) $f(x)$ is false, then there is no need to lift above that sector/section, as it is automatically false (and other equivalent notions).
- Shortcuts for determining signs of polynomials in certain cells.

Partial CAD has largely been overtaken by methods that also improve the projection, described later in this section.

2.8.3 Block Ordering and Block Cylindricity

CAD implementations typically assume a *total* variable order, but this need not always be the case. For a quantified prenex Tarski formula Φ of the form (2.4), variables within the same quantifier block commute, as do free variables, and their order within a block can be chosen freely [Bro04]. Thus, Φ can be expressed as

$$\Phi = Q_1(\mathbf{x}_1) \dots Q_q(\mathbf{x}_q) \phi(\mathbf{y}_F, \mathbf{x}_1, \dots, \mathbf{x}_q), \quad (2.5)$$

where $\mathbf{x}_i$ denotes the set of variables quantified by Q_i and $\mathbf{y}_F$ the set of free variables.

We make use of the notions of *block ordering* and *block cylindricity* [Dav], in which the variable ordering is specified only between blocks, and the associated decompositions are cylindrical with respect to this block ordering. A block containing the unordered variables $\mathbf{x}$ is denoted $[\mathbf{x}]$, and the block ordering $w \prec [\mathbf{x}] \prec y$ denotes that the variables in $\mathbf{x}$ are eliminated, in some order, after y and before w . Using this notation, a QE problem in the form (2.5) induces the ordering:

$$[\mathbf{y}_F] \prec [\mathbf{x}_1] \prec \dots \prec [\mathbf{x}_q].$$

This block ordering notation is useful when only a partial order is known or required, allowing for a guided choice of variable ordering while still allowing for flexibility. This idea is expanded upon in Chapter 3, in which this additional freedom is explored.

2.8.4 Open CAD

In the previous section, we have discussed CAD construction: repeatedly projecting a set of polynomials $\mathcal{F}_n \subset \mathbb{R}[x_1, \dots, x_n]$ to obtain a full collection of projection polynomials $\mathcal{P} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$, where each $\mathcal{F}_i \subset \mathbb{R}[x_1, \dots, x_i]$. Starting with $\mathcal{F}_1$, we decompose $\mathbb{R}$ into sections (the roots of the polynomials of $\mathcal{F}_1$) and sectors (the intervals either side of them). A sample point from each cell is lifted into the polynomials in $\mathcal{F}_2$, the space above each cell is again decomposed, and the process repeats until a CAD of $\mathbb{R}^n$ is obtained.

This construction produces cells of varying dimension: the decomposition of $\mathbb{R}$ produces 0-cells (sections) and 1-cells (sectors). In general, the cylinder above an i -cell C is decomposed into i -cells (sections) and $i+1$ -cells (sectors), and so the final CAD of $\mathbb{R}^n$ consists of cells of dimension $0 \leq i \leq n$, where i represents the number of variables that are not fixed, i.e. the number of sector cells in or below the cell in question.

Recall from Remark 2.3.3 the field required for CAD construction: our polynomials are assumed to have coefficients in $\mathbb{Q}$ (or equivalently $\mathbb{Z}$), and this remains true for every polynomial in $\mathcal{P}$. It is only during the root isolation and lifting that the field

$\mathbb{A}$ is required. The roots of univariate polynomials that arise during this process are often expressed in terms of the polynomial and a rational isolating interval¹. Storing, tracking or even describing such roots can be complicated, and full implementation of a CAD algorithm is not possible without a way to do so.

Despite this, there are many QE problems in which algebraic numbers are not technically required. For example, for existential problems involving strict inequalities, only full-dimensional n -cells are relevant: the formula is automatically false for any other i -cell in the CAD of $\mathbb{R}^n$. In these n -cells (and in fact in any cell below them) one may always pick a rational sample point in $\mathbb{Q}$, which may not be true for cells on or above a section.

This leads to the notion of an *open CAD*: a subset of a *full CAD*, which consists only of open, full-dimensional n -cells, and is sufficient for specialised cases. The advantages are twofold: open CADs can still be produced while remaining purely in $\mathbb{Q}$, allowing open CAD algorithms to be implemented in environments where full CADs cannot, and for cases in which an open CAD is sufficient, constructing one instead of its full CAD equivalent would produce considerably fewer cells, leading to a considerable performance improvement. One additional bonus of an open CAD is that the sample points for each open cell lie in $\mathbb{Q}^n$, and this representation may be more meaningful to a potential user.

In many applications, such open cells capture the essential geometry of a problem, while closed cells often represent special cases, making open CADs applicable and an attractive alternative to full CADs in those situations. An implementation of an open CAD algorithm is described in Chapter 4.

2.8.5 Equational Constraints

A powerful tool for reducing the complexity of CAD computation is the use of equational constraints (ECs), equations logically implied by the quantifier-free part of the formula. If an EC is present, it suffices to decompose the variety defined by the EC instead of the entirety of $\mathbb{R}^n$, usually reducing the double exponent of the number of polynomials in the CAD complexity bound by one. More generally, if multiple ECs can be exploited consecutively, the problem becomes that of decomposing a space of relatively small dimension and this double exponent is reduced by one for each subsequent EC used, assuming they are algebraically independent [BDE⁺16, EBD20, DNSU23].

¹For example, *Maple*'s RootOf notation, in which roots are stored as either 'the root of the (univariate) polynomial f in this interval' or 'the k th distinct root of f (ordered from lowest to highest)', and in works such as [Col96], in which roots are formally and informally described by a polynomial and their index or interval.

Definition 2.8.4 (Equational constraint). *Given a Tarski formula Φ , an equational constraint is a polynomial equation logically implied by the quantifier-free formula ϕ .*

An equational constraint is said to be explicit if it is an atomic formula connected by logical conjunction, otherwise it is implicit. That is, if one can write

$$\phi = \phi^* \wedge (g = 0) \quad (2.6)$$

then $g = 0$ is an explicit EC, and g is called the equational constraint polynomial.

Some example of implicit ECs include:

- The formula $f = 0 \vee g = 0$ has $fg = 0$ as an implicit EC.
- The formula $f^2 + g^2 \leq 0$ has $f = 0$ and $g = 0$ as implicit ECs.
- The formula $(f^2 + g^2 \leq 1) \wedge ((f - 2)^2 + g^2 \leq 1)$ has $f = 1$ and $g = 0$ as implicit ECs.

In general, one means an explicit EC when referring to an EC. In practice, only explicit ECs will be known and exploitable by algorithms, but implicit ECs may result in additional simplification if identified. One implicit EC of note arises via the *resultant rule*, that is, $(f = 0 \wedge g = 0) \Rightarrow \text{res}(f, g) = 0$.

The notion of an equational constraint was first proposed by Collins [Col98], who observed that if an EC is present, it can be used to reduce the projection sets required for CAD-based QE. McCallum [McC99] formalised this, giving a detailed proof for a *restricted projection* which makes use of an EC, and later introduced a weaker semi-restricted projection operator which could take advantage of subsequent ECs [McC01]. More recently, these restricted projections have been applied to more contemporary projections [DNSU23].

The principle behind exploiting an EC of the form (2.6) is that the formula can only be true if both ϕ^* and $g = 0$ are true: if $g \neq 0$, the formula is false, and if $g = 0$, its truth depends on ϕ^* . Therefore, it suffices to decompose the variety V_g rather than all of $\mathbb{R}^n$, as the formula is known to be false elsewhere, making additional computation there unnecessary.

When a formula contains multiple ECs, one is chosen as the *pivot*, with the remaining ECs treated like any other polynomial. The pivot can be a product of several irreducible factors, as its discriminant will contain the discriminants and resultants of its factors. Consequently, obtaining the implicit EC $fg = 0$ from $f = 0 \vee g = 0$ can be valuable, as disjunctions of multiple ECs can be exploited simultaneously, whereas conjunctions cannot.

Restricted Projections

Let us consider this more formally: suppose the input formula is of the form (2.6) and contains an EC $g = 0$. Since the formula is automatically false outside V_g , no further decomposition is required there, as the truth value is already determined. As a result, our projection operator can omit polynomials which do not involve g , namely the coefficients and discriminants of, and resultants exclusively between, the other polynomials.

In practice, we let $\mathcal{F} \subset \mathbb{R}[x_1, \dots, x_n]$ be a set of pairwise irreducible factors of polynomials, and let $\mathcal{G} \subset \mathcal{F}$ be the set of irreducible factors of the pivot. The *restricted projection of $\mathcal{F}$ with respect to $\mathcal{G}$* is given by

$$\text{proj}_{\mathcal{G}}(\mathcal{F}) := \text{proj}(\mathcal{G}) \cup \text{res}(\mathcal{G}, \mathcal{F} \setminus \mathcal{G}).$$

As in Theorem 2.6.4, to make use of a projection proj and a valuation ν , we need the following to hold:

Theorem 2.8.5 (Restricted Lifting Theorem). *Let $\mathcal{F} = \{f_1, \dots, f_r\}$ be a set of pairwise coprime irreducible polynomials (an irreducible basis) in $n \geq 2$ variables $x_1, \dots, x_n$ of positive degrees in x_n and $\mathcal{G}$ a subset of $\mathcal{F}$. Let S be a connected subset of $\mathbb{R}^{n-1}$ on which each element of $\text{proj}_{\mathcal{G}}(\mathcal{F})$ is ν -invariant. Then each element of $\mathcal{G}$ is ν -delineable over S and each element of $\mathcal{G}$ is ν -invariant in the pairwise disjoint ν -sections of $\mathcal{G}$ over S . Each element of $\mathcal{F} \setminus \mathcal{G}$ is **sign**-invariant in each such ν -section.*

We give two examples of this:

- the *restricted McCallum projection* [McC99]: $\text{PM}_{\mathcal{G}}(\mathcal{F}) = \text{PM}(\mathcal{G}) \cup \text{res}(\mathcal{G}, \mathcal{F} \setminus \mathcal{G})$,
- the *restricted Lazard projection* [Nai21]: $\text{PL}_{\mathcal{G}}(\mathcal{F}) = \text{PL}(\mathcal{G}) \cup \text{res}(\mathcal{G}, \mathcal{F} \setminus \mathcal{G})$,

where PM and PL are the McCallum projection and the Lazard projection respectively, see Section 2.6.1. As with the unrestricted McCallum projection $\text{PM}(\mathcal{F})$, $\text{PM}_{\mathcal{G}}(\mathcal{F})$ will not satisfy Theorem 2.8.5 when the polynomials used have curtains above S . Additionally, $\text{PL}_{\mathcal{G}}(\mathcal{F})$ can also fail on curtains under certain circumstances [Nai21] unlike its unrestricted equivalent.

Note that Theorem 2.8.5 only guarantees sign-invariance, not ν -invariance, and therefore can only be used to replace the first and last projections. If this restricted projection is used for the first projection, then the last lifting step involves lifting a ν -invariant decomposition of $\mathbb{R}^{n-1}$ to a decomposition of $\mathbb{R}^n$ in which the input polynomials are sign-invariant on V_g . Similarly, it holds for the final projection, as sign-invariance

guarantees ν -invariance for the univariate projection polynomials on the ν -sections: any region of non-zero valuation is a point and thus has a fixed valuation.

If one is able to make use of this restricted projection, it offers a significant reduction in complexity due to the creation of fewer cells, but this fails if the EC nullifies anywhere, or if nullification causes a problem with the lift to $\mathbb{R}^{n-1}$.

Semi-Restricted Projections

The restricted projection has a strong advantage in terms of complexity, but cannot be used iteratively, even if more ECs were available. In order to exploit multiple ECs we need a projection operator which ensures the lifting polynomials are ν -invariant, allowing for ECs to be used at every level. We introduce the *semi-restricted* projection operator:

$$\begin{aligned}\text{proj}_{\mathcal{G}}^*(\mathcal{F}) &:= \text{proj}_{\mathcal{G}}(\mathcal{F}) \cup \text{disc}(\mathcal{F} \setminus \mathcal{G}) \cup \text{projterms}(\mathcal{F} \setminus \mathcal{G}), \\ &= \text{disc}(\mathcal{F}) \cup \text{projterms}(\mathcal{F}) \cup \text{res}(\mathcal{F}, \mathcal{G}), \\ &= \text{proj}(\mathcal{F}) \setminus \text{res}(\mathcal{F} \setminus \mathcal{G}, \mathcal{F} \setminus \mathcal{G}).\end{aligned}$$

As before, to make use of a projection proj and a valuation ν , we need the following to hold:

Theorem 2.8.6 (Semi-restricted Lifting Theorem). *Let $\mathcal{F} = \{f_1, \dots, f_r\}$ be a set of pairwise relatively prime irreducible polynomials (an irreducible basis) in $n \geq 2$ variables $x_1, \dots, x_n$ of positive degrees in x_n and $\mathcal{G}$ a subset of $\mathcal{F}$. Let S be a connected subset of $\mathbb{R}^{n-1}$ on which each element of $\text{proj}_{\mathcal{G}}^*(\mathcal{F})$ is ν -invariant. Then each element of $\mathcal{G}$ is ν -delineable over S and each element of $\mathcal{G}$ is ν -invariant on the pairwise disjoint ν -sections of $\mathcal{G}$ over S .*

Our two examples are the *semi-restricted McCallum projection* [McC01] and the *semi-restricted Lazard projection* [Nai21], given by

$$\begin{aligned}\text{PM}_{\mathcal{G}}^*(\mathcal{F}) &:= \text{PM}_{\mathcal{G}}(\mathcal{F}) \cup \text{disc}(\mathcal{F} \setminus \mathcal{G}) \cup \text{coeffs}(\mathcal{F} \setminus \mathcal{G}), \\ \text{PL}_{\mathcal{G}}^*(\mathcal{F}) &:= \text{PL}_{\mathcal{G}}(\mathcal{F}) \cup \text{disc}(\mathcal{F} \setminus \mathcal{G}) \cup \text{lc}(\mathcal{F} \setminus \mathcal{G}) \cup \text{tc}(\mathcal{F} \setminus \mathcal{G})\end{aligned}$$

respectively. Similarly to the restricted case, $\text{PM}_{\mathcal{G}}^*(\mathcal{F})$ fails in the presence of curtains, while $\text{PL}_{\mathcal{G}}^*(\mathcal{F})$ fails for some curtains but works (with a minor modification) for others.

If multiple ECs are present, one may use the restricted projection for the first and last projections, and the semi-restricted projection for each intermediate projection. Naturally, the semi-restricted projection can only be used if there exists an EC at a

lower level. One way of ensuring this is through *EC propagation* using the resultant rule: any resultants of ECs are ECs themselves, meaning any $\text{res}(\mathcal{G}, \mathcal{G})$ and any $\text{res}(\mathcal{G}, \mathcal{F} \setminus \mathcal{G})$ between ECs can be marked as ECs and used at lower levels.

While propagation is one way to ensure ECs at lower levels, using repeated resultants as pivots may produce spurious roots [EBD20]. Another way to ensure ECs at lower levels is by producing a Gröbner basis of the ECs of $\mathcal{F}$, producing a new set of ECs with the same common roots and disjoint main variables. This ensures that fewer projection polynomials are computed at higher levels and ECs can be used at lower levels without propagation. Like CAD itself, Gröbner basis calculation is known to be doubly-exponential in the worst case [MM82], but can often be done relatively quickly.

2.8.6 Truth-Table-Invariance

A CAD produced using restricted or semi-restricted projection is no longer sign- or ν -invariant for a set of polynomials, but truth-invariant for the formula. We have seen that the truth-invariance of a formula Φ can be determined from a ν -invariant CAD with respect to $\mathcal{F}$, the set of polynomials appearing in Φ , and have shown that restricted projection can be used when ECs are present in the formula to produce a decomposition with fewer cells. We now extend this idea further.

If our input is a basic semi-algebraic set $\mathcal{F}_\diamond = \{f_i \diamond_i 0\}$ for each $f_i \in \mathbb{R}[x_1, \dots, x_n]$, its associated formula is the conjunction of these constraints, and restricted projection may be used if any of these constraints are ECs.

More generally, any quantifier-free Tarski formula Φ may be built from conjunctions or disjunctions of sub-formulae, and its truth value may be determined recursively: if $\Phi = \phi_1 \wedge \phi_2$, then Φ is false when either sub-formula is false, and if $\Phi = \phi_1 \vee \phi_2$, then Φ is true when either sub-formula is true.

Recall that any quantifier-free Tarski formula Φ can be written in DNF (2.3), meaning Φ is true if any conjunction is true, and if a conjunction is false, the truth of Φ depends on the remaining conjunctions. This is a generalisation of the theory of equational constraints, as in order to determine the truth value for Φ , one must first determine the truth value of (enough of) its conjunctions, each of which may contain an EC.

We have noted that a basic semi-algebraic set is the solution set of a conjunction, and that a semi-algebraic set is the solution set of a DNF quantifier-free Tarski formula. Our input here is a list of quantifier-free formulae whose truth depends on these formulae. This list typically represents a disjunction, so if Φ is a disjunction of sub-formulae, the input is a list of the semi-algebraic sets associated to these sub-formulae. If Φ is a

conjunction of sub-formulae, each sub-formula is split into a separate input, and this is repeated until a list of semi-algebraic sets is obtained. Note that a list does not have to correspond to a DNF Tarski formula (in which case it would be written as a list of conjunctions, or as a list of basic semi-algebraic sets representing these conjunctions), and writing a quantifier-free Tarski formula in DNF may not always be beneficial¹.

Definition 2.8.7 (Truth-table-invariant [BDE⁺13]). *Let $\Phi = \{\phi_i\}_{i=1}^t$ be a list of quantifier-free formulae ϕ_i . A CAD is truth-table-invariant (a TTICAD) for Φ if each ϕ_i has constant truth value on each cell of the CAD.*

In short: there may be cases where an input formula itself does not contain an EC, but its sub-formulae do, and a TTICAD is able to take advantage of these ECs via restricted projections.

An algorithm, written in terms of the McCallum projection PM (see Section 2.6.1), for directly building TTICADs was first introduced in [BDE⁺13] for cases where each ϕ_i contains an EC. It was extended in [BDE⁺16] for arbitrary ϕ_i , with savings if at least one of the ϕ_i contains an EC.

Definition 2.8.8 (Reduced projection (TTICAD)). *Let $\mathbf{F}$ be a list of sets $\mathcal{F}_{(i)}$ and $\mathbf{G}$ a list of sets $\mathcal{G}_{(i)}$ such that each $\mathcal{G}_{(i)} \subseteq \mathcal{F}_{(i)} \subset \mathbb{R}[x_1, \dots, x_n]$. Then the reduced projection of $\mathbf{F}$ with respect to $\mathbf{G}$ is given by*

$$\text{PM}_{\mathbf{G}}(\mathbf{F}) := \bigcup_{i=1}^k \text{PM}_{\mathcal{G}_{(i)}}(\mathcal{F}_{(i)}) \cup \text{Res}^\times(\mathbf{G}),$$

where

$$\text{Res}^\times(\mathbf{G}) := \text{res}_{x_n}(\mathcal{G}_{(i)}, \mathcal{G}_{(j)}), \text{ for } i < j.$$

One could instead compute the restricted projection $\text{PM}_{\bigcup \mathcal{G}_{(i)}}(\bigcup \mathcal{F}_{(i)})$ (that is, a restricted projection of the terms in every $\mathcal{F}_{(i)}$ with respect to the product of every pivot), but this reduced projection uses fewer polynomials in general.

At present, this reduced projection operator appears to exist only in a form analogous to the fully restricted projection (see Section 2.8.5), and only for the McCallum projection. As such, it may only be used in similar circumstances, i.e. for the first and last projections, and when not in the presence of curtains. Extending TTICAD to include an equivalent to semi-restricted projection, and to support the Lazard projection operator, is a worthwhile area of future research.

¹For example, consider the formula $\Phi = \phi_1 \wedge \phi_2 \wedge (\phi_3 \vee \phi_4)$, which in DNF becomes $(\phi_1 \wedge \phi_2 \wedge \phi_3) \vee (\phi_1 \wedge \phi_2 \wedge \phi_4)$. Using the latter means only one input is required, but this involves many more sub-formulas, some of which are repeated, while the former formula avoids this.

We conclude this subsection by giving the associated lifting theorem:

Theorem 2.8.9 (Reduced Lifting Theorem [BDE⁺16]). *Let S be a connected submanifold of $\mathbb{R}^{n-1}$. Suppose each element of $\text{PM}_{\mathbf{G}}(\mathbf{F})$ is order-invariant in S . Then each $g \in \bigcup \mathcal{G}_{(i)}$ either vanishes identically on S or is analytically delineable over S ; the sections over S of the $g \in \bigcup \mathcal{G}_{(i)}$ which do not vanish identically are pairwise disjoint; and each element $g \in \bigcup \mathcal{G}_{(i)}$ which does not vanish identically is order-invariant in such sections.*

Moreover, for each i , in $1 \leq i \leq t$ every $f \in \mathcal{F}_{(i)} \setminus \mathcal{G}_{(i)}$ is sign-invariant in each section over S of every $g \in \mathcal{G}_{(i)}$ which does not vanish identically.

2.9 CAD via Regular Chains

In [CMMXY09, CMM14a] an alternative to the projection and lifting method for computing CADs was introduced. This approach is based on the theory of triangular sets and regular chains, and first computes a *cylindrical decomposition* of $\mathbb{C}^n$ (CCD), which is then refined into a CAD through real root isolation.

The initial recursive algorithm of [CMMXY09] offered advantages over projection and lifting methods, avoiding redundant computations and the problem of curtains. However, it involved ‘many black boxes, which hide many unnecessary or redundant computations’ [CMM14a], and often required more computation time than projection and lifting methods, despite usually producing fewer cells.

In [CMM14a], the authors instead presented an incremental algorithm for CCD construction, in which the associated *complex cylindrical tree* is refined with respect to each input polynomial using GCD computations. This has been shown to be competitive with projection and lifting algorithms, and will be a particular focus in what follows and in Chapters 5 and 6.

Another benefit of this incremental approach is its natural handling of equational constraints. The algorithm separates input polynomials into equations and inequations, so when these are provided, this separation and the redundant branch can be avoided. This reduces both computation time and output size, with [CMM14a] evidencing examples where this optimisation made problems tractable that other implementations could not.

Remark 2.9.1. *As a complex analogue to Remark 2.3.3, throughout this section we write $f \in \mathbb{C}[x_1, \dots, x_n]$ to denote a polynomial with rational coefficients. Complex roots*

of polynomial equations lie in the field of complex algebraic numbers $\mathbb{A}_{\mathbb{C}} = \overline{\mathbb{Q}}$, but in the context below this notation is not needed. While the method for constructing a complex cylindrical tree always ensures the associated polynomials have rational coefficients and no roots are explicitly displayed (merely represented as the solution set of $f = 0$ or $f \neq 0$), this notation can still be a useful reminder.

2.9.1 Triangular Sets and Regular Chains

Definition 2.9.2 (Triangular set). Let $\mathcal{T} = \{t_1, \dots, t_s\}$ be a set of polynomials in $\mathbb{C}[x_1, \dots, x_n]$. We say $\mathcal{T}$ is a triangular set if its polynomials are non-constant and have pairwise distinct main variables.

Definition 2.9.3 (Regular chain). Let $\mathcal{T} = \{t_1, \dots, t_s\}$ be a triangular set with the variable ordering $\text{mvar}(t_1) \prec \dots \prec \text{mvar}(t_s)$, and let $h_{\mathcal{T}}$ be the product of the leading coefficients of its polynomials. We say $\mathcal{T}$ is a regular chain if either $\mathcal{T} = \emptyset$ or the iterated resultant

$$\text{ires}(h_{\mathcal{T}}, \mathcal{T}) = \text{res}_{\text{mvar}(t_1)}(\dots (\text{res}_{\text{mvar}(t_{s-1})}(\text{res}_{\text{mvar}(t_s)}(h_{\mathcal{T}}, t_s), t_{s-1}) \dots), t_1)$$

is non-zero.

Informally, a regular chain is a triangular set of polynomials whose leading coefficients do not vanish at the roots of lower-level polynomials.

Definition 2.9.4 (Squarefree triangular set). Denote by $\text{der}(f)$ the derivative of f with respect to its main variable. The triangular set $\mathcal{T}$ is said to be squarefree if $\text{ires}(\prod_{f \in \mathcal{T}} \text{der}(f), \mathcal{T}) \neq 0$.

Note that this definition is consistent with that in Section 2.5.4: a polynomial with repeated roots will share those roots with its derivative, and thus its discriminant will be zero. Recall that for any polynomial f , one can produce a squarefree polynomial $\hat{f} = \text{sf}(f)$ with the same roots using the GCD of f and its derivative.

Definition 2.9.5 (Simple system [Wan98]). A simple system is a pair of disjoint polynomial sets $[T, \hat{T}]$ such that $T \cup \hat{T}$ is a squarefree regular chain.

2.9.2 Complex Equivalents

We now look at extending familiar terminology to the complex domain.

Definition 2.9.6 (Separation). Let $S \subseteq \mathbb{C}^{n-1}$ and let $\mathcal{F} \subset \mathbb{C}[x_1, \dots, x_n]$ be a finite set of level n polynomials. We say that $\mathcal{F}$ separates above S if for each $\sigma \in S$:

- for each $f \in \mathcal{F}$, $\text{lc}(f)$ does not vanish at σ ,
- the polynomials $f(\sigma, x_n) \in \mathbb{C}[x_n]$, for all $f \in \mathcal{F}$, are squarefree and coprime.

Two such polynomials $f(\sigma, x_n)$ and $g(\sigma, x_n)$ are coprime if and only if they never vanish simultaneously, i.e. their zero sets are disjoint.

Definition 2.9.7 (Invariance). *Let $\mathcal{F}$ be a finite set of polynomials of $\mathbb{C}[x_1, \dots, x_n]$ and $S \subseteq \mathbb{C}^n$. We say $f \in \mathcal{F}$ is (sign-)invariant¹ on S if it is either identically zero or never zero on S , and S is $\mathcal{F}$ -invariant if each $f \in \mathcal{F}$ is invariant on S .*

Definition 2.9.8 (Cylindrical decomposition of $\mathbb{C}^n$ and associated tree). *By induction on n , we define the notion of a cylindrical decomposition of $\mathbb{C}^n$ (CCD).*

For $n = 1$, a cylindrical decomposition of $\mathbb{C}$ consists of a finite collection of sets $\mathcal{D} = \{D_1, \dots, D_{r+1}\}$, where either $r = 0$ and $D_1 = \mathbb{C}$, or $r > 0$ and there exists r nonconstant coprime squarefree polynomials $f_1, \dots, f_r$ of $\mathbb{C}[x_1]$ such that for $1 \leq i \leq r$ we have

$$D_i = \{x_1 \in \mathbb{C} \mid f_i(x_1) = 0\}, \quad \text{and} \quad D_{r+1} = \{x_1 \in \mathbb{C} \mid f_1(x_1) \cdots f_r(x_1) \neq 0\}.$$

For $n > 1$, let $\mathcal{D}' = \{D_1, \dots, D_s\}$ be any cylindrical decomposition of $\mathbb{C}^{n-1}$. For each D_i , let $\{f_{i,1}, \dots, f_{i,r_i}\}$ be a set of polynomials which separates above D_i , for $r \geq 0$. If $r_i = 0$, set $D_{i,1} = D_i \times \mathbb{C}$. If $r_i > 0$, then for $1 \leq j \leq r_i$ we have

$$D_{i,j} = \{(\sigma, x_n) \in \mathbb{C}^n \mid \sigma \in D_i \text{ and } f_{i,j}(\sigma, x_n) = 0\}$$

$$D_{i,r_i+1} = \{(\sigma, x_n) \in \mathbb{C}^n \mid \sigma \in D_i \text{ and } \prod_{j=1}^{r_i} f_{i,j}(\sigma, x_n) \neq 0\}.$$

The collection $\mathcal{D} = \{D_{i,j} \mid 1 \leq i \leq s, \ 1 \leq j \leq r_i + 1\}$ is called a cylindrical decomposition of $\mathbb{C}^n$. The sets $D_{i,j}$ are called the cells of $\mathcal{D}$.

A cylindrical decomposition $\mathcal{D}$ of $\mathbb{C}^n$ is $\mathcal{F}$ -invariant if every cell D of $\mathcal{D}$ is $\mathcal{F}$ -invariant. As with CAD, we will use $\text{CD}(\mathcal{F})$ to refer to a cylindrical decomposition of $\mathbb{C}^n$ that is invariant with respect to $\mathcal{F}$.

This CCD can be seen as the complex analogue of a CAD, with the cells $D_{i,j}$ and D_{i,r_i+1} analogous to the sections and sectors, with the important distinction that in the complex setting, the roots of a polynomial are treated collectively, as the concept of distinct ordering or separation of roots has no meaning in $\mathbb{C}$.

¹This is the complex equivalent of sign-invariance as given in Definition 2.5.5, with the notion of ‘sign’ in $\mathbb{C}$ simply meaning ‘zero’ or ‘non-zero’.

Definition 2.9.9 (Solution set over $\mathbb{C}$). *For a polynomial $f \in \mathbb{C}[x_1, \dots, x_n]$, we denote by $Z_{\mathbb{C}}(f \star 0) \subseteq \mathbb{C}^n$ the solution set² of the constraint $f \star 0$, for $\star \in \{=, \neq\}$, and denote the zero set of f by $Z_{\mathbb{C}}(f = 0)$. Trivially, $Z_{\mathbb{C}}(f) = Z_{\mathbb{C}}(f = 0) \cup Z_{\mathbb{C}}(f \neq 0) = \mathbb{C}^n$.*

Each cell of a CCD is either a zero set of a level n polynomial or the solution set of the inequation involving their product.

2.9.3 Complex Cylindrical Trees

We also consider the *complex cylindrical tree* T associated with a cylindrical decomposition of $\mathbb{C}^n$. Such a tree T is a rooted tree of depth n , whose nodes of depth k are associated with the polynomial constraint $f = 0$ or $f \neq 0$, for some polynomial f of level k , or the trivial constraint ‘any x_k ’. The depth of a node is the length of the path between the node and the root.

The polynomial constraints along a path from root to leaf form a polynomial system called a *cylindrical system of $\mathbb{C}[x_1, \dots, x_n]$ induced by T* . Every cylindrical system induced by a CCT is a simple system whose sets consist of equations and inequations.

We denote by Γ a path of T and define its solution set $Z_{\mathbb{C}}(\Gamma)$ as the intersection of solution sets of its nodes. The solution set of T , denoted $Z_{\mathbb{C}}(T)$, is the union of the solution sets of its paths. Each cell of a CCD $\mathcal{D}$ is the solution set of a cylindrical system induced by T .

This CCT T has a recursive definition which parallels that of a cylindrical decomposition given in Definition 2.9.8: each cell of the CCD is the solution set of the polynomial constraint on a leaf node of the CCT at a point σ in the solution set of the polynomial constraint of the parent node (with the trivial cell $D_i \times \mathbb{C}$ corresponding to ‘any x_i ’). More explicitly, each path Γ is a cylindrical system, and each cell of $\mathcal{D}$ is its solution set. The associated CCD is given by $Z_{\mathbb{C}}(T) = \{Z_{\mathbb{C}}(\Gamma) \mid \Gamma \text{ is a path of } T\} = \mathbb{C}^n$.

It is useful to discuss the notion of invariance with regard to a CCT.

Definition 2.9.10 (Invertible/zero/coprime modulo Γ). *Let Γ be a cylindrical system of $\mathbb{C}[x_1, \dots, x_n]$. A polynomial $f \in \mathbb{C}[x_1, \dots, x_n]$ is invertible modulo Γ or zero modulo Γ if $f(\sigma) \neq 0$ or $f(\sigma) = 0$, respectively, for every $\sigma \in Z_{\mathbb{C}}(\Gamma)$. For $g \in \mathbb{C}[x_1, \dots, x_n]$, we say f and g are coprime modulo Γ if their zero sets (inside $Z_{\mathbb{C}}(\Gamma)$) are disjoint.*

The polynomial f is invariant on Γ if it is either invertible modulo Γ or zero modulo Γ . A constraint $f = 0$ or $f \neq 0$ is truth-invariant on Γ if f is invariant on Γ .

²In [CMM14a] ‘zero set’ is used throughout, and much of the following terminology is expressed in terms of zeroes of polynomials. We instead use ‘solution set’ to emphasise the connection between the polynomials and the cells of the CCD.

Definition 2.9.11 (GCD mod Γ). *Let f and g be level n polynomials in $\mathbb{C}[x_1, \dots, x_n]$ and Γ a cylindrical system of $\mathbb{C}[x_1, \dots, x_{n-1}]$. Assume that $\text{lc}(f(\sigma, x_n))$ and $\text{lc}(g(\sigma, x_n))$ are coprime modulo Γ . A polynomial $G \in \mathbb{C}[x_1, \dots, x_{n-1}][x_n]$ is a GCD of f and g modulo Γ if, for any such σ , $G(\sigma, x_n) \in \mathbb{C}[x_n]$ is a GCD of $f(\sigma, x_n)$ and $g(\sigma, x_n)$, and $\text{lc}(G)$ is invertible modulo Γ .*

Recall from Section 2.5.3 the relationship between polynomial GCDs and subresultants. For f and g two level n polynomials in $\mathbb{C}[x_1, \dots, x_n]$, Γ a cylindrical system of $\mathbb{C}[x_1, \dots, x_{n-1}]$, and G the GCD of f and g modulo Γ , it is clear that f and g are coprime modulo Γ if $\deg_{x_n}(G) = 0$.

Let f and g be two level n polynomials in $\mathbb{C}[x_1, \dots, x_n]$ with $\deg(f) \geq d := \deg(g) \geq 1$ and let Γ be a cylindrical system of $\mathbb{C}[x_1, \dots, x_{n-1}]$. Set $S_d = g$, $S_{d+1} = f$, $s_d = \text{lc}(g)$ and $s_{d+1} = \text{lc}(f)$, and let S_i and s_i for $0 \leq i \leq d-1$ denote the i th subresultants and principal subresultant coefficients for f and g respectively.

Theorem 2.9.12 (Subresultants and GCD mod Γ). *Let j be an integer, $1 \leq j \leq d+1$, such that s_j is invertible modulo Γ and such that for any $0 \leq i < j$, we have s_i is zero modulo Γ . Then S_j is a GCD of f and g modulo Γ .*

Note that this is the equivalent of Theorem 2.5.16.

2.9.4 The Incremental Algorithm

With these definitions, we now describe the workings of the refinement algorithm. For a more in-depth explanation, including each sub-algorithm, see [CMM14a].

This algorithm takes a set of polynomials $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{C}[x_1, \dots, x_n]$ and constructs the CCT associated with the $\mathcal{F}$ -invariant CCD $\text{CD}(\mathcal{F})$. Starting by setting T as the *initial tree*, a tree containing one node at each level with the constraints ‘any x_k ’ (associated to the trivial decomposition $\text{CD}(\emptyset)$ containing only the cell $\mathbb{C}^n$), we *refine* T incrementally with respect to each polynomial $f_i \in \mathcal{F}$ using GCD calculation and separation methods (see Section 2.5.4), so that the associated cylindrical decomposition is invariant with respect to f_i in addition to any previous polynomials. This continues until T has been refined with respect to every polynomial in $\mathcal{F}$. Each path of T is a cylindrical system, and its solution set is the associated cell of the $\mathcal{F}$ -invariant CCD.

This refinement works as follows. Let f be a non-constant polynomial in $\mathcal{F}$ of level k . For each node N_k of level k :

1. If the formula on N_k is ‘*any* x_k ’, it is split into the two nodes $\text{sf}(f) = 0$ and $\text{sf}(f) \neq 0$.

If the formula on N_k is $g \star 0$, the GCD of $\text{sf}(f)$ and g is computed and GCD separation is used to give $G, \hat{f} = \text{sf}(f)/G, \hat{g} = g/G$. Then:

2. A node with the formula $g = 0$ is split into $G = 0$ and $\hat{g} = 0$.
3. A node with the formula $g \neq 0$ is split into $\hat{f} = 0$ and $g\hat{f} \neq 0$.

The GCDs produced to compute $\text{sf}(f), \hat{f}$ and $\hat{g}$ are subresultants, and the PSCs $\text{disc}(f)$ and $\text{res}(f, g)$ obtained during this are recursively used to split lower-level formulae.

The final step is to compute $\text{CAD}(\mathcal{F})$ from $\text{CD}(\mathcal{F})$. This is constructed such that for each cell $D \in \text{CD}(\mathcal{F})$, the set $D \cap \mathbb{R}^n$ is a union of cells in $\text{CAD}(\mathcal{F})$. Indeed, when D is a cell of $\text{CD}(\mathcal{F})$ associated with a level n polynomial equation², $D \cap \mathbb{R}^n$ is naturally a union of section cells of $\text{CAD}(\mathcal{F})$, and for D associated with an polynomial inequation of level n , $D \cap \mathbb{R}^n$ is a union of sector cells.

Producing a CAD from the CCD is now a matter of standard stack construction and lifting, with the polynomial equations of level i playing the part of the projection polynomials $\mathcal{F}_i$ seen in projection and lifting CAD algorithms.

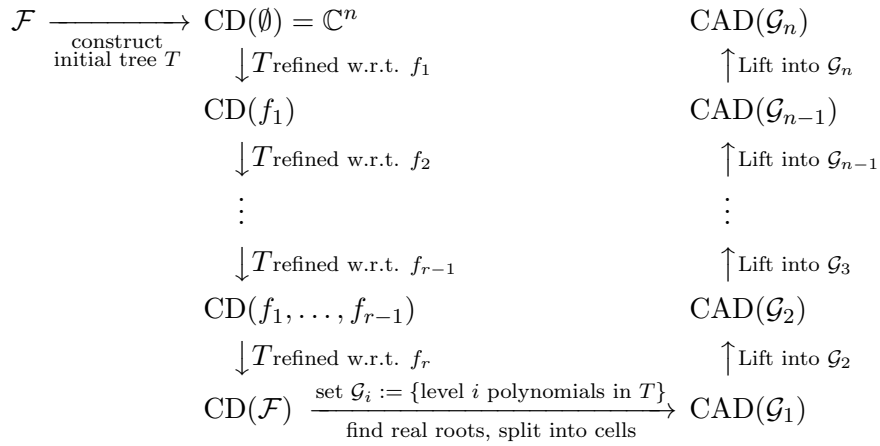


Figure 2-6: An overview of the incremental algorithm. Note that the steps on the left increment by polynomial, while those on the right increment by dimension, meaning each CCD on the left has no direct relation to the CAD to its right.

²i.e., each $D_{i,j}$ for $1 \leq j \leq r_i$ as given in Definition 2.9.8.

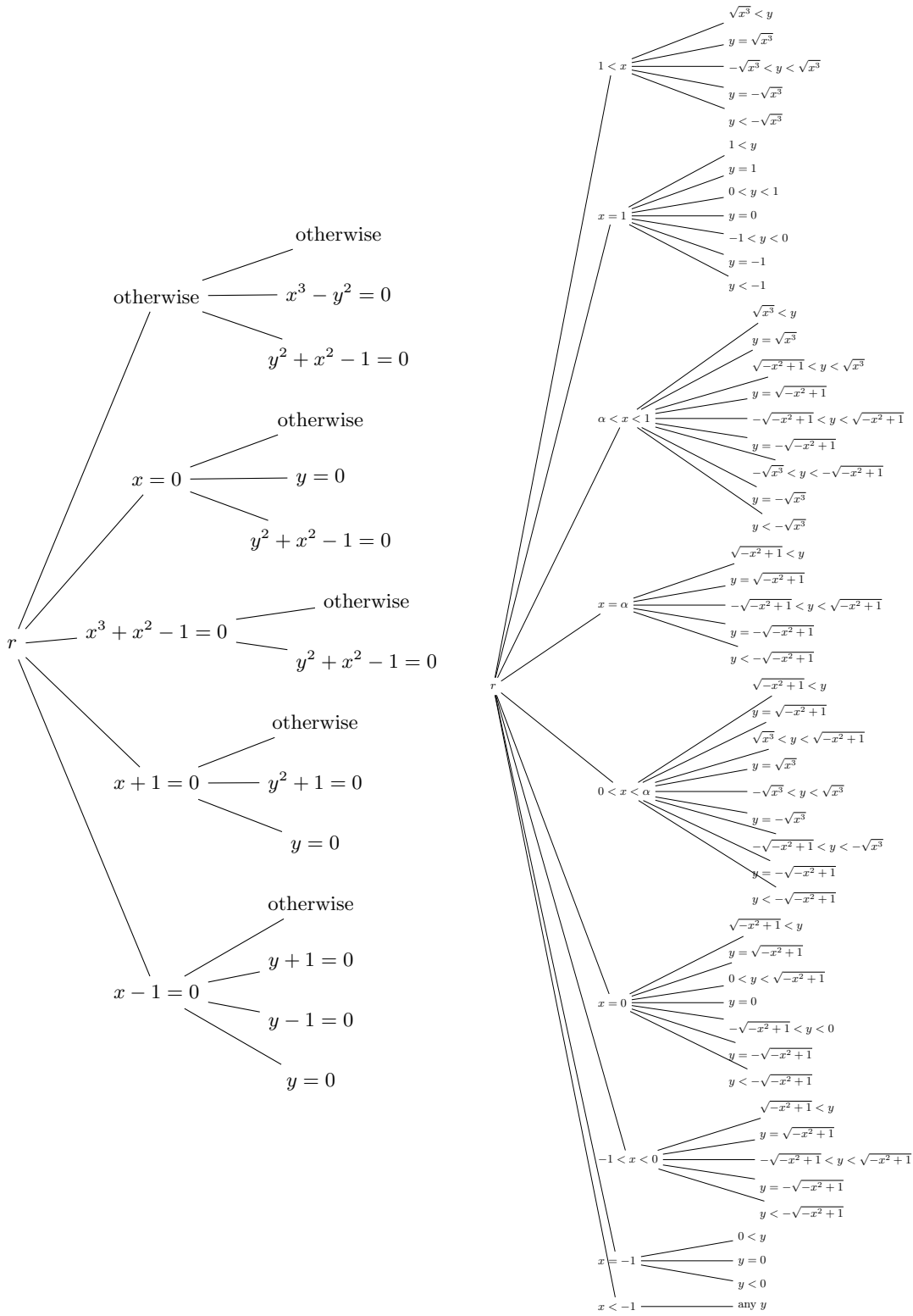


Figure 2-7: A CCD of $\{x^2 + y^2 - 1, x^3 - y^2\}$ and the resulting CAD, for the variable ordering $x < y$. Here α is the real root of $x^3 + x^2 - 1 = 0$.

2.9.5 Regular Chains with Equational Constraints

This incremental algorithm is naturally well suited to handling equational constraints. A key step in the algorithm involves using a polynomial to split a node into a polynomial equation and inequation, and if the input is already one of these constraints, the other is irrelevant and its branch need not be constructed.

If presented with a collection of equations or inequations, the algorithm will compute a partial cylindrical tree whose zero set is exactly the zero set of the input system, but we must consider how to translate any set of real polynomial constraints to complex ones.

Definition 2.9.13 (Complex system). *Let $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{C}[x_1, \dots, x_n]$ and $\mathcal{G} \subseteq \mathcal{F}$. A basic complex system $\mathcal{F}_\star \subset \mathbb{C}^n$ is the set*

$$\mathcal{F}_\star := \mathcal{G}_\star \cup (\mathcal{F} \setminus \mathcal{G}) = \{f_i \star_i 0 \mid f_i \in \mathcal{G}\} \cup \{f_i \mid f_i \in \mathcal{F} \setminus \mathcal{G}\}, \quad (2.7)$$

where $\star_i \in \{=, \neq\}$. When $\mathcal{F} \setminus \mathcal{G} = \emptyset$, we call $\mathcal{F}_\star$ complete, and partial otherwise. Note that a complete basic complex system is a cylindrical system.

A basic complex system is formed from a (basic) semi-algebraic system $\{f_i \diamond_i 0\}$ with $\diamond_i \in \{>, =, \geq, \neq\}$ by replacing each $f_i > 0$ by $f_i \neq 0$ and each $f_i \geq 0$ by f_i .

We define a complex system¹ as a union of basic complex systems, and call it complete if each such basic complex system is complete, and partial otherwise.

Refining with respect to a polynomial constraint allows for significant simplification. For example, if we refine T with respect to the equation $f = 0$, then any branches in which f is known to be non-zero are removed, and computations are skipped over branches above which f cannot be zero.

If given a constraint $f \star 0$, we can skip computing $\text{sf}(f)$ and simply use f in each case (as the relevant polynomials will still vanish at the same points) and can immediately mark $\text{res}(f, g)$ computed during GCD calculation as zero.

¹Note that in [BCD⁺14], ‘complex system’ is instead used to refer to a basic complex system, and a collection of such sets (our notion of a complex system) is stored as a list.

Remark 2.9.14. *The three refinement cases given in Section 2.9.4 simplify to the following when presented with a polynomial constraint.*

1. Splitting ‘any x_k ’ into $\text{sf}(f) = 0$ and $\text{sf}(f) \neq 0$:
 - we only need to compute the branch $f \star 0$.
2. Splitting $g = 0$ into $G = 0$ and $\hat{g} = 0$:
 - for $f = 0$, we only need to compute the branch $G = 0$.
 - for $f \neq 0$, we only need to compute the branch $\hat{g} = 0$.
3. Splitting $g \neq 0$ into $\hat{f} = 0$ and $g\hat{f} \neq 0$:
 - for $f = 0$, we only need to compute the branch $\hat{f} = 0$.
 - for $f \neq 0$, we only need to compute the branch $g\hat{f} \neq 0$.

Additionally, for the node formula $g \star 0$, g can be replaced by $\text{pp}(g)$, as $\text{lc}(g)$ is invertible modulo the truncated path. Replacing g by its irreducible factors (over $\mathbb{Q}$) is often a more efficient choice.

Let us now consider when the input is not just a set of constraints.

Every polynomial f can be split into $(f \neq 0) \vee (f = 0)$, and every (partial) basic complex system $\mathcal{F}_\star = \{f_1 \star_1 0, \dots, f_s \star_s 0, g\}$ can be written as an equivalent complete complex system

$$\{f_1 \star_1 0, \dots, f_s \star_s 0, g \neq 0\} \cup \{f_1 \star_1 0, \dots, f_s \star_s 0, g = 0\}.$$

Naturally the CCT refinement process can handle a partial basic complex system as input, and the CCT produced is a complete complex system, a union of the cylindrical systems associated to the paths of the tree.

Remark 2.9.15. *As with the notion of a basic semi-algebraic system (Definition 2.5.3), a complete basic complex system $\mathcal{F}_\star$ is the set in which a conjunction of constraints $\bigwedge (f_i \star_i 0)$ is true, for each $f_i \in \mathcal{F} \subset \mathbb{C}[x_1, \dots, x_n]$.*

This notion is less useful when considering a partial basic complex system: this is a set in which multiple conjunctions of constraints are true (i.e. the formula obtained by replacing each $f \in \mathcal{F} \setminus \mathcal{G}$ with $(f = 0) \vee (f \neq 0)$ and writing this in DNF). While this system is also the set in which $\bigwedge (g_i \star_i 0)$ is true, for $g_i \in \mathcal{G} \subset \mathcal{F}$ (as any element of $\mathcal{F} \setminus \mathcal{G}$ has no constraint in this basic complex system and so is true everywhere), we avoid any implication that the remaining elements can be ignored or discarded.

One can construct a CCT invariant with respect to a complex system by producing a CCT for each constituent basic complex system, then merging their root nodes. This will involve redundant computation if the basic complex systems share elements, so it may be more worthwhile to instead ‘flatten’ this complex system into a weaker basic partial complex system and compute its tree. In contrast, producing the equivalent DNF-like complete complex system would require producing many CCTs with many duplicated branches.

2.9.6 Regular Chains and Truth-Table-Invariance

Let us now look at constructing a CCT and CAD that are truth-invariant with respect to an input formula. When a CCT T is refined with respect to a polynomial constraint $f \star 0$, T is refined so the constraint is true, and the branch in which it is false is truncated.

Definition 2.9.16 (Truth-invariance on Γ). *Let Γ be a path of a CCT T and let $f \in \mathbb{C}[x_1, \dots, x_n]$ be a polynomial. A constraint $f \star 0$ is truth-invariant on Γ if f is invariant on Γ . A complete basic complex system is truth-invariant on Γ if the conjunction of the polynomial constraints is truth-invariant on Γ , and each polynomial is invariant on Γ .*

As mentioned in Remark 2.9.15, a basic complex system (2.7) is true precisely when the conjunction of its elements is true, and in fact its truth value only depends on the truth values of its polynomial constraints. Recall that a polynomial f in a basic complex system is obtained from the real constraint $f \leq 0$, and not distinguishing cases for this polynomial means losing a way to distinguish signs when the CCD produced is later refined into a CAD, so these unconstrained polynomials cannot be discarded.

We now revisit the three possible refinement cases given in Remark 2.9.14 with truth-invariance in mind. Note that $\text{sf}(f)$ and f will have the same truth values everywhere as they contain the same set of distinct roots, and so computing $\text{sf}(f)$ can be skipped. Additionally, if $f \neq 0$, then the GCD of f and g is invertible, and so the truth values for $\hat{f}$ and $\hat{g}$ are the same as those of f and g , so one could skip GCD separation and use f and g instead.

If $\mathcal{F}_\star$ contains a polynomial constraint $f \star 0$, the set is false if $f \star 0$ is false, and if $f \star 0$ is true, the truth of the $\mathcal{F}_\star$ relies on the truth of $\mathcal{F}'_\star = \mathcal{F}_\star \setminus \{f \star 0\}$. This is also clear with regard to the cases in Remark 2.9.14: if passed a constraint $f \star 0$, the truth values are unchanged when this constraint is true, and the other case is known to be false. In terms of solution sets, under a constraint $f \star 0$ we have $\{Z_{\mathbb{C}}(f = 0), Z_{\mathbb{C}}(f \neq 0)\} = \{\mathbb{C}^n, \emptyset\}$.

In a similar sense to the projection and lifting algorithm of Section 2.8.6, constructing a TTICAD using this incremental algorithm requires a list (union) of basic complex systems as input. This does not directly map to DNF, as a polynomial f represents the disjunction $(f = 0) \vee (f \neq 0)$, but otherwise can be seen as analogous to it.

Definition 2.9.17 (Truth-table-invariant CCT/CCD). *Let $\mathcal{CS}$ be a complex system of polynomials $f_i \in \mathcal{F} \subset \mathbb{C}[x_1, \dots, x_n]$ and let T be a CCT with paths Γ_j . The complex system $\mathcal{CS}$ is truth-table-invariant on T (a TTICCT) if $\mathcal{CS}$ can be written as a union of basic complex systems, each of which is truth-invariant on every path Γ_j of T . The associated CCD is truth-table-invariant (a TTICCD) for an appropriate quantifier-free Tarski formula Φ , which is true precisely in $\mathcal{CS}$.*

As the construction of CCTs and CCDs depends on the roots of the polynomials, ECs are a natural focus. When refining a CCT to be truth-invariant with respect to the complex system $\mathcal{CS}$ containing a set $\mathcal{F}_\star$ with an EC $f = 0$, savings can be made depending on the truth of the EC. On branches where $f = 0$ is true, we need only ensure that $(\mathcal{CS} \setminus \mathcal{F}_\star) \cup \mathcal{F}'_\star$ is truth-invariant. On branches where $f = 0$ is false, we know $\mathcal{F}_\star$ is false, and it suffices to check the truth value of $\mathcal{CS} \setminus \mathcal{F}_\star$.

In the case that $\mathcal{CS}$ contains no ECs (i.e. each set contains only unconstrained polynomials and inequations), one could consider solving each separately (by refining each tree to be sign-invariant with respect to the polynomials when the inequations are all true), but this may not always be better. As this is a union of these sets, the inequations in one set do not need to be true if another set is true, so one can simply use the polynomials in $\mathcal{F}$ to produce a (sign-)invariant CCT.

We are able to obtain a truth-table-invariant CCT from a basic complex system, obtained from a (basic) semi-algebraic set, and the same is true for a complex system. Recall that an EC is unchanged when obtaining $\mathcal{F}_\star$ from $\mathcal{F}_\diamond$ and that the CCD is a union of cells D . Each $D \cap \mathbb{R}^n$ is a union of cells C in the CAD, so any CAD produced from the truth-table-invariant CCD is also a TTICAD.

In short: To produce a TTICAD for an unquantified Tarski formula Φ , we first obtain a semi-algebraic system associated to it, convert this to a complex system, and handle this as in Section 2.8.6, separating higher-level conjunctions and treating disjunctions as lists, until each input is a list of basic complex systems. If Φ is in DNF, this will result in a single list. We then look to refine a CCT T with respect to this list of basic complex systems. In this list $\mathcal{CS}$, we consider a basic complex system $\mathcal{F}_\star$ with an EC $f = 0$ and look to refine T with respect to f . On branches in which $f = 0$ is true,

we then look to refine with respect to the same list with $f = 0$ removed from $\mathcal{F}_*$, and on branches where it is false, we look to refine with respect to the same list but with $\mathcal{F}_*$ removed, repeating this recursively until we have our TTICCT and the associated TTICCD, from which we may obtain our TTICAD of Φ .

This is similar to the idea behind projection and lifting TTICAD, but with the savings gained by using case distinction and multiple ECs in each element of the list, and with the further benefit of not needing to worry about curtains.

Chapter 3

A Case Study: The Solotareff Approximation Problem

3.1 Introduction

Solotareff's first problem [Ach56, p.280], or the Solotareff approximation problem, looks to find the best (i.e. the closest approximation in the uniform norm) degree $n - 2$ approximation $A_n(x)$ of the polynomial $f(x) = x^n + rx^{n-1}$ in the interval $[-1, 1]$ for a given $r \geq 0$. This is a generalisation of a problem by Chebyshev [Ach56, p.57], which seeks the monic degree n polynomial which deviates the least from zero in an interval, the solution of which depends on the Chebyshev polynomial $T_n(x)$.

The best approximation $A_n(x)$ in Solotareff's problem is uniquely characterised by Chebyshev's theorem: the error function $E_n(x) = f(x) - A_n(x)$ attains its maximum absolute value at no fewer than n *maximal deviation points* u_i in $[-1, 1]$.

Collins [Col96] reformulated Solotareff's problem as a quantifier elimination problem and applied cylindrical algebraic decomposition (CAD) using QEPCAD. He aimed to compute the coefficients of $A_n(x)$ as piecewise algebraic functions of r for small fixed n . He succeeded fully for $n \leq 4$, and partially for $n = 5$, using the tool's interactive features alongside intuition, in order to avoid the 'impractically large' computation time that would otherwise arise.

In many cases, the CAD algorithm requires a fixed variable ordering, and this can obscure further simplifications. We show that by eliminating certain unquantified variables first, and exploiting equational constraints, inequalities, and symmetries, the

problem can be reduced significantly. In the $n = 5$ case, these tools allowed us to express new variables as functions of r without using QEPCAD, highlighting that these tools can uncover features that a naive, purely algorithmic approach might overlook.

The aim of this chapter is not to solve Solotareff’s problem in general—explicit algebraic solutions for $n \leq 12$ have already been obtained by Rack and Vajda [RV19] using Malyshev polynomials. Rather, as with Collins’ original approach using QEPCAD, our goal is to use Solotareff’s problem as a test case: a well-structured yet non-trivial example to illustrate and evaluate new strategies for quantifier elimination and CAD.

As demonstrated later in this chapter, these techniques can yield significant improvements. For example, while Collins was only able to partially solve the case $n = 5$, our approach allows it to be solved almost entirely by hand. This offers some compelling evidence that such methods are worth pursuing further.

3.2 A Toolbox of Strategies

We present a collection of techniques that help push back the ‘doubly-exponential wall’ inherent in CAD computation, allowing us to tackle problems that would otherwise be intractable. While the worst-case complexity remains doubly-exponential in the number of variables, even taking doubly-exponential time to write the output down [DH88], careful application of these strategies can make a significant difference in practice.

Because of this complexity, CAD works best on problems with a small number of variables and polynomials of lower degree. What follows is a collection of tricks involving variable ordering, symmetry, equational constraints and degree reduction, which can push CAD a step or two further than expected.

3.2.1 Leveraging Equations and Inequalities

As mentioned in Section 2.8.5, one powerful and well-studied technique for reducing the complexity of CAD computation is the use of equational constraints (ECs), equations logically implied by the quantifier-free part of the formula. By exploiting ECs, one can decompose a smaller solution space, leading to significant performance improvements.

ECs are particularly useful in the Solotareff problem: the formulation given in Section 3.3.1 includes a large number of ECs—one fewer than the number of variables. The presence of multiple ECs permits this restriction to continue at lower levels, with additional ECs found via substitution or EC propagation, enabling further variable elimination and reduction in complexity.

Geometrically, the conjunction of these ECs defines the common variety of their polynomials, providing insight into the dimension of the solution set. If the polynomials define a complete intersection, this variety would be a curve, but this cannot be assumed *a priori*. Additionally, implicit ECs arising from inequalities may further reduce the dimension.

In this case, however, Chebyshev’s theorem guarantees that the best approximation polynomial $A_n(x)$ is uniquely determined by, and varies continuously with, the parameter r . As a result, the solution set is a curve parametrised by r , which will be given by the ECs.

Some of the techniques described below are similar to the use of ECs, Gröbner bases, and Virtual Term Substitution [Wei88], reducing complexity by simplifying the structure of the problem. Additionally, features such as symmetries and inequalities can be lost when key variables are eliminated, so choosing a variable ordering that preserves these features allows for more opportunities to exploit them.

3.2.2 Buchberger-Type Elimination Using Equational Constraints

When presented with multiple ECs, it is often well worth pre-conditioning the associated polynomials by replacing them with a Gröbner basis, which contains the same roots and is triangular, meaning resultants between polynomials are not required for projection or EC propagation.

The following process is similar to Buchberger’s algorithm for computing Gröbner bases, but is performed explicitly through substitution of leading terms.

We can eliminate a variable X from a system of multivariate polynomial constraints through iterative substitution if it appears in at least as many equations as its lowest degree among those equations, and those equations are linearly independent over the polynomial ring in the remaining variables.

That is, if X appears in d equational constraints $\mathbf{E}_1, \dots, \mathbf{E}_d$, for $d = \min \deg_X(\mathbf{E}_i)$, we can use an equation of degree d to reduce the remaining equations and lower the X -degree of the system. By repeating this process, we can obtain an equation linear in X and thus a solution that eliminates the variable.

Without loss of generality, suppose $\mathbf{E}_d$ has degree d , so can be written as:

$$\mathbf{E}_d : \quad \alpha X^d + f(X) = 0, \quad \text{with } \deg_X(f) < d \text{ and } \alpha \neq 0.$$

We then isolate the leading term:

$$X^d = -\frac{f(X)}{\alpha}.$$

Each other constraint $\mathbf{E}_i$, for $1 \leq i < d$, can expressed similarly:

$$\mathbf{E}_i : \quad \alpha_i X^{d_i} + f_i(X) = 0, \quad \text{with } \deg_X(f_i) < d_i.$$

Since $d_i \geq d$, we can substitute the above value of X^d into each leading term:

$$X^{d_i} = X^{q_i d + r_i} = (X^d)^{q_i} X^{r_i} = \left(-\frac{f(X)}{\alpha}\right)^{q_i} X^{r_i}, \quad \text{where } 0 \leq r_i < d.$$

Note here that α is a polynomial in the remaining variables, so making this substitution can lower the degree, but at the cost of introducing rational function coefficients. There may also be points at which $\alpha = 0$, where one will need to set up *guards* à la Virtual Term Substitution. Such cases will need to be treated separately, but this will provide an additional degree drop when the leading terms of all the degree d polynomials vanish simultaneously.

However, if α is a non-zero constant, then the substitution ensures that coefficients are still polynomials in the other variables, and variables are eliminated. This is mostly the case in the example we discuss later.

By repeatedly substituting X^d in this way across all terms with degree at least d , we reduce each equation to degree strictly less than d , giving a new system, namely $\mathbf{E}_d$ and the other reduced equations:

$$\mathbf{E}'_i : \quad \alpha'_i X^{d'_i} + g_i(X) = 0, \quad \text{with } \deg_X(g_i) < d'_i < d.$$

We then repeat the process using an equation of new minimal degree d' , continuing until we obtain an equation linear in X , from which the variable can be eliminated.

3.2.3 Main Tools

We now outline the key tools used in our simplification strategy. In the worked example later, the tools are used together to achieve a significant improvement, but each can be used independently, and a given problem may have any number of the features used by these tools.

- Block cylindricity (see Section 3.2.4): We can group certain variables into a block, leaving their order within the block unspecified while preserving their order relative to other variables. This provides added flexibility when projecting and eliminating variables, and may preserve features that would otherwise be obfuscated by certain orderings.
- Symmetry exploitation (see Section 3.2.5): Some problems contain symmetries between variables, which can be leveraged by expressing constraints in terms of invariants, leading to simplification through degree or term reduction.
- Effective quantification (see Section 3.2.6): Quantifier elimination focuses on bound variables, but sometimes some free variables can be treated as bound, allowing for early elimination. This can lead to more efficient elimination orders that would otherwise be overlooked, especially when these variables appear in low degree and in multiple equational constraints, and when there are more equational constraints than bound variables.
- Exploiting inequalities (see Section 3.3.3): When a problem contains multiple ordered points (such as the maximal deviation points given by Chebyshev's theorem), fixing an explicit ordering gives rise to known inequalities and thus known non-zero factors which can be eliminated from constraints, leading to further complexity reductions.

3.2.4 Block Cylindricity

The formulation of QE problems naturally imposes an order on the variables: quantified variables must be ordered above free ones, and in the order specified by the quantifier prefix.

Given a Tarski formula:

$$\Phi = Q_k x_k Q_{k+1} x_{k+1} \dots Q_n x_n \phi(x_1, \dots, x_n), \quad (3.1)$$

where $Q_i \in \{\exists, \forall\}$, CAD implementations would naturally eliminate the quantified variables in the order $x_n, x_{n-1}, \dots, x_k$, then eliminate the free variables until one re-

mained. However, variables within the same quantifier block (i.e., adjacent, and bound by the same quantifier) are commutable and may be eliminated in any order within the block [Bro04], meaning Φ can be expressed as:

$$\Phi = Q_1(\mathbf{x}_1) \dots Q_q(\mathbf{x}_q) \phi(\mathbf{y}_F, \mathbf{x}_1, \dots, \mathbf{x}_q)$$

where $\mathbf{x}_i$ denotes the set of variables quantified by Q_i and $\mathbf{y}_F$ the set of free variables.

Despite this inherent partial order, most implementations of QE via CAD instead either require a specified total ordering, or produce one implied by the quantifier prefix of (3.1). Such an ordering can be chosen without consideration of the quantifier-free part ϕ , and may therefore be poorly adapted to the structure of the problem.

The need for a total order in these implementations means that, when several variables are equally suitable for projection or when their ordering depends on other parameters, a user cannot ask the algorithm to determine an order among only those variables, and would instead have to determine this another way. An algorithmic implementation of partial ordering would address this, allowing variables within a block to be ordered either heuristically or through user intervention.

We further expand the notion of *block ordering* and *block cylindricity* introduced in Section 2.8.3, in which the variable ordering is specified only between blocks, and the associated decompositions are cylindrical with respect to this block ordering.

Recall that for a set of variables $\mathbf{x} = x_1, \dots, x_n$, we denote by $[\mathbf{x}]$ the unordered block of variables in $\mathbf{x}$, and the block ordering $w \prec [\mathbf{x}] \prec y$ specifies that these variables are eliminated, in some order, after y and before w . We denote by $\prec_{\mathbf{x}}$ an (unspecified) ordering among the elements of $\mathbf{x}$, and note that this block ordering is equivalent to

$$w \prec \bigvee_{\sigma \in \mathcal{S}_n} \prec_{\{x_{\sigma(i)}\}} \prec y,$$

where $\mathcal{S}_n$ is the symmetric group on n elements.

We are in fact permitted to make invertible linear changes of variables among the elements in the block, so equivalently, we have

$$w \prec \bigvee_{M \in \text{GL}_n} \prec_{\{M(x_i)\}} \prec y,$$

where GL_n is the general linear group of invertible $n \times n$ matrices.

The ability to manipulate the variables and their ordering provides greater flexibility and allows one to identify a new ordering better suited to the structure of ϕ , such as choosing variables that make the equations effectively triangular, allowing for Gaussian elimination.

While some algorithms allow for the fixing of the first few variables through the quantifier prefix, consider many different orderings via a greedy heuristic [DSS04, Ton21], or allow limited interaction by allowing the user to abort or continue when presented with the current projection [Bro], no existing CAD implementations appear to support user-specified partial variable orderings. The **GraphCAD** package (see Chapter 6) addresses this by allowing the user to specify certain variables be eliminated either before or after all others, and in the order given, with the other variables ordered using the Brown heuristic.

3.2.5 Symmetric Polynomials

A polynomial is *symmetric* in the $m \leq n$ variables of $\hat{\mathbf{x}} \subseteq \mathbf{x}$ if it remains unchanged under any permutation of these variables. That is, it is invariant under the action of S_m on the variables in $\hat{\mathbf{x}}$.

More generally, polynomials which are invariant under a finite group acting on its variables can be written in terms of the generators of the corresponding invariant subring, which can simplify the system. As a result, it can be beneficial to choose a variable ordering that delays the elimination of the variables involved, in order to preserve and exploit this invariance.

When the polynomials are symmetric, the invariant subring is generated by the *elementary symmetric polynomials* $X_1, \dots, X_m$, where for each $1 \leq a_1 < a_2 < \dots < a_k \leq n$,

$$X_k(\hat{x}_1, \dots, \hat{x}_m) = \sum_{a_1, \dots, a_k} \prod_{j=1}^k \hat{x}_{a_j}, \quad \text{for } 1 \leq k \leq m.$$

Rewriting symmetric polynomials in terms of the elementary symmetric polynomials typically reduces both degree and the number of terms, making this an attractive property one should try to retain.

In many cases, a system of polynomials may only exhibit symmetry among its equations. In such cases, it can still be worthwhile to rewrite these symmetric equations in terms of invariants in order to simplify and solve the system where possible, using back-substitution to return to the original variables when they are required for inequalities.

3.2.6 Effective Quantification

In cases where a quantified formula contains more ECs than quantified variables, it may also be possible to solve for and eliminate some of the nominally free variables. While heuristics suggest eliminating lower degree variables first, CAD-based QE tools such as QEPCAD will eliminate quantified variables before free ones, and without considering opportunities for early simplification. This can result in a suboptimal elimination order and can significantly affect performance.

We introduce the concept of *effective quantification*: in such cases, certain free variables may be uniquely determined by the ECs and as a result can be treated as existentially quantified (as indeed, they have known solutions). We call such variables *effectively quantified*.

Effective quantification is another tool that looks at the quantifier-free formula and allows one to consider variable orderings which would otherwise be ignored. Quantifying additional variables allows lower-degree variables to be eliminated earlier whilst postponing the elimination of those associated with symmetries and useful inequalities, which can lead to significant improvement.

3.2.7 Takeaways

The situations in which these tools apply are not isolated special cases. Considerable work has been undertaken on using ECs to reduce the complexity of CAD problems, and these additional tools may help push the wall back further. They rely on analysing the structure of the input formula to identify useful features and construct a more effective variable ordering with those features in mind.

Concepts such as block cylindricity and effective quantification offer flexibility not available in current algorithmic implementations, allowing one to consider the problem structure and features such as inequalities and symmetries, something which would not be possible under an algorithmically determined total order depending only on the logical structure. By deviating from the standard algorithmic approach, and considering other options, one can preserve or exploit features that would be lost otherwise, leading to significant improvements.

This is demonstrated in the final section of this chapter, where many of these tools are used together to turn a previously intractable problem into one that is largely solvable by hand.

3.3 Chebyshev's Theorem

The Sotolareff approximation problem [Ach56, p.280] asks for the monic, degree n polynomial with given second coefficient, that deviates least from zero in $[-1, 1]$. This is equivalent to finding the best approximating polynomial to a degree n monic polynomial in the space of degree $n - 2$ polynomials. Such a polynomial is proven to exist, and is uniquely characterised by specific properties, thanks to the following special case of Chebyshev's theorem.

Theorem 3.3.1 (Chebyshev's Theorem, special case [Ach56, p.57]). *If $f(x)$ is a real-valued function continuous on the closed bounded interval $[a, b]$, then the degree n polynomial $A(x)$ which deviates less from $f(x)$ than any other degree n polynomial $B(x)$ in $[a, b]$ is uniquely characterised by having $N \geq n + 2$ maximal deviation points $u_1 < \dots < u_N$ in the interval $[a, b]$ at which the difference $|f(x) - A(x)|$ assumes the value $L = \max_{a \leq x \leq b} |f(x) - A(x)|$. This difference $f(x) - A(x)$ alternates in sign at each successive point.*

For the Sotolareff problem, we have¹

- $f(x) = x^n + rx^{n-1}$ for a given non-negative r ,
- $A_n(x) = a_{n-2}x^{n-2} + \dots + a_0$ the best approximation polynomial in the interval $[-1, 1]$ among the space of degree $n - 2$ polynomials,
- $E_n(x) = f(x) - A_n(x)$ their difference.

Chebyshev's theorem tells us that $A_n(x)$ is uniquely characterised by the existence of $N \geq n$ maximal deviation points $-1 \leq u_1 \leq \dots \leq u_N \leq 1$, with the properties that $|E_n(u_i)| = \|E_n\|_\infty = L$ and $E_n(u_i) = (-1)^{i-1}E_n(u_1)$.

The best approximation can look very different depending on the value of r chosen. For $0 \leq r \leq S_n = n(\tan(\frac{\pi}{2n}))^2$, the error $E_n(x)$ has a lower form which is easily computable.

¹The original lower form presented by Achieser was for $f(x) = x^n - n\sigma x^{n-1}$, for $\sigma \geq 0$. Collins uses the relation $E_n(x) = (-1)^n E_n(-x)$ to obtain an approximation for $x^n + rx^{n-1}$ from the approximation of $x^n - rx^{n-1}$, leading to Equation 3.2. To avoid unnecessary confusion, we will use the form presented by Collins throughout.

This lower form is given by

$$E_n(x) = \frac{(-1)^n}{2^{n-1}} (1 + r/n)^n T_n \left(\frac{-x - r/n}{1 + r/n} \right), \quad (3.2)$$

where $T_n(x) = \cos(n \arccos x)$ is the *Chebyshev polynomial of degree n* , and $L = (1/2)^{n-1}(1 + r/n)^n$. This formula gives the coefficients of $A_n(x)$ as polynomial functions of r .

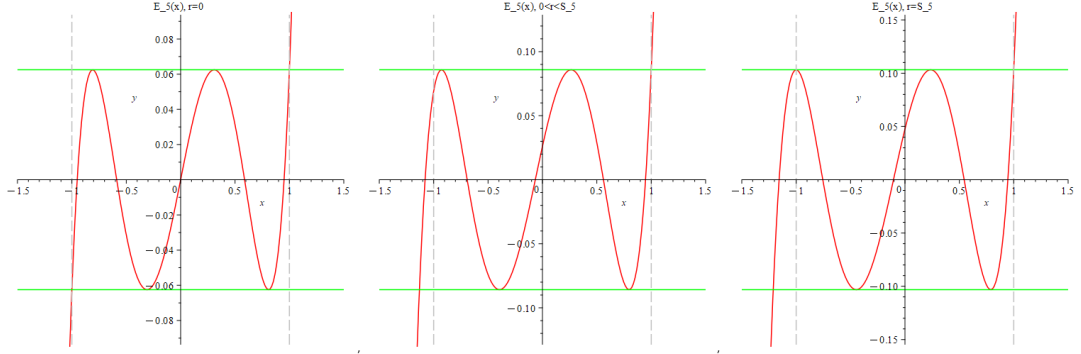


Figure 3-1: The graph of the lower form of $E_5(x)$, for different values of r . The green lines are $\pm L$.

Remark 3.3.2. *Note some features of this lower form:*

- By Chebyshev's theorem, there must exist at least n maximal deviation points, i.e. distinct roots of $E_n(x)^2 - L^2$, in $[-1, 1]$.
- When $r = 0$, the right-hand side of Equation 3.2 becomes $L \cdot T_n(x)$, which has $n + 1$ maximal deviation points at $x = -\cos(\pi k/n)$, for $0 \leq k \leq n$.
- When $0 < r \leq S_n$, there are exactly n maximal deviation points, with $u_n = 1$ always holding and giving $E_n(1) = L$.
- $x = 1$ is a simple root of $E_n(x)^2 - L^2$, and the remaining maximal deviation points are double roots. Between each maximal deviation point is a root of $E_n(x)$, meaning $E_n(x)$ has n real roots.
- When $r = S_n$, we also have $u_1 = -1$, with $E_n(-1) = (-1)^n L$.

Once $r > S_n$, this formula no longer yields the best approximation, as it has fewer than n maximal deviation points in the interval $[-1, 1]$, so a new best approximation must be found. This *higher form* is not so easy to compute, but the following is known.

Remark 3.3.3. *This higher form of $E_n(x)$ must have the following properties:*

1. *By Chebyshev's theorem, there are N maximal deviation points in $[-1, 1]$, and the value of $E_n(x)$ alternates in sign at each successive point.*
2. *The coefficients of $A_n(x)$ are continuous functions of r for $r > S_n$ [Col96], and so $E_n(x)$ is continuous for all $r \geq 0$. Therefore, it also holds for the higher form that $N = n$, $u_1 = -1$ and $u_n = 1$, with $E_n(1) = L$ and $E_n(-1) = (-1)^n L$.*
3. *We have $E_n(1) = L > 0$, so $x = 1$ is a simple root of $E_n(x)^2 - L^2$, and $u_2, \dots, u_{n-1}$ are double roots and turning points.*
4. *As $E_n(-1) = (-1)^n L$, there must be another turning point for $x < -1$ and a root of E_n at some $\gamma < -1$. Thus $x = -1$ is a simple root, and $E_n(x)^2 - L^2$ has two more simple roots at α and β , where $\alpha < \gamma < \beta < -1$, with $E_n(\beta) = E_n(-1)$ and $E_n(\alpha) = -E_n(-1)$.*
5. *The sum of the roots of $E_n(x)$ is $-r$, and since $n - 1$ of these are bounded by the interval $[-1, 1]$, we must have $\gamma \rightarrow -\infty$ as $r \rightarrow \infty$.*

Achieser notes that $E_n(x)^2 - L^2 \leq 0$ in the intervals $[\alpha, \beta]$ and $[-1, 1]$, and considers a function on the complex x -plane slit along these intervals, using elliptic functions in order to express $E_n(x)$ in terms of Green's function, determining L and verifying that the polynomial is determined for all $r > S_n$.

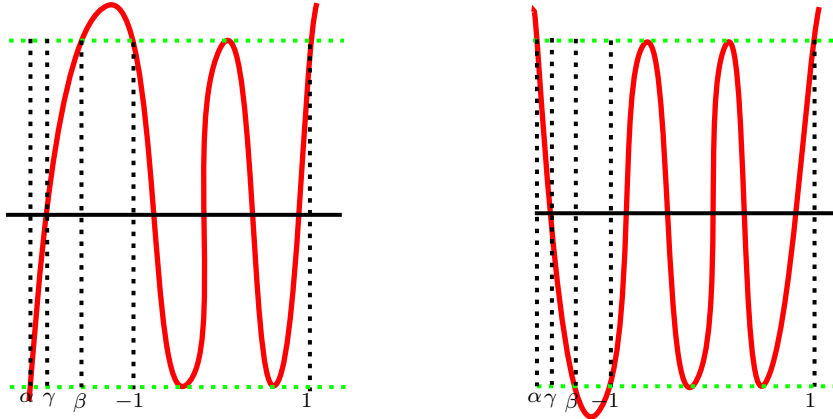


Figure 3-2: A rough sketch of the higher form of $E_n(x)$ for $r > S_n$, when n is odd (left) and when n is even (right).

Following Collins [Col96, p.183], we will instead look at the higher form as a QE problem:

3.3.1 Collins' Formulation

To express the Solotareff problem in these terms, for any polynomial $B_n(x)$ of degree $n - 2$, we write $F_n(x) = f(x) - B_n(x)$. The condition for $A_n(x)$ to be a better uniform approximation to $f(x)$ than $B_n(x)$ in $[-1, 1]$ can be written as:

$$\|E_n\|_\infty \leq \|F_n\|_\infty \Leftrightarrow (\forall x)[(-1 \leq x \leq 1) \Rightarrow (\exists y)[(-1 \leq y \leq 1) \wedge |E_n(x)| \leq |F_n(y)|]]. \quad (3.3)$$

Note that $|E_n(x)| \leq |F_n(y)|$ is equivalent to

$$[(F_n(y) \geq 0) \wedge (-F_n(y) \leq E_n(x) \leq F_n(y))] \vee [(F_n(y) \leq 0) \wedge (F_n(y) \leq E_n(x) \leq -F_n(y))].$$

For $f(x) = x^n + rx^{n-1}$ with best approximation $A_n(x) = a_{n-2}x^{n-2} + \dots + a_0$ and any competing approximation $B_n(x) = b_{n-2}x^{n-2} + \dots + b_0$, this can be written as:

$$(\forall b_0, \dots, b_{n-2}, x) \quad [(-1 \leq x \leq 1) \Rightarrow (\exists y)[(-1 \leq y \leq 1) \wedge |f(x) - A_n(x)| \leq |f(y) - B_n(y)|]] \quad (3.4)$$

This quantified formula has n free variables $(r, a_{n-2}, \dots, a_0)$ and $n + 1$ bound variables $(x, y, b_{n-2}, \dots, b_0)$, and incrementing n by one adds one free and one bound variable. However, this is a large number of variables and a large number of quantifiers to eliminate. Collins uses Chebyshev's theorem and the properties in Remark 3.3.3 to reduce the number of quantified variables.

There are n maximal deviation points at which $E_n(x)$ alternates in sign, and by continuity we have that $u_1 = -1$, $u_n = 1$ and $E_n(1) = L$, meaning that $E_n(u_i) = (-1)^{n-i}E_n(1)$ for $-1 = u_1 < u_2 < \dots < u_{n-1} < u_n = 1$. As u_2 to u_{n-1} are turning points and double roots of $E_n(x)^2 - L^2$, we also have that $E'_n(u_i) = 0$.

By Chebyshev's Theorem, these properties characterise $E_n(x)$, giving the following quantified formula:

$$\begin{aligned} (\exists u_2, \dots, \exists u_{n-1}) \quad & [(r > S_n) \wedge (-1 < u_2 < \dots < u_{n-1} < 1) \wedge (E_n(1) > 0) \\ & \wedge (E_n(-1) = (-1)^{n-1}E_n(1)) \\ & \wedge_{i=2}^{n-1} [E_n(u_i) = (-1)^{n-i}E_n(1) \wedge E'_n(u_i) = 0]] \end{aligned} \quad (3.5)$$

This formula involves the same n free variables $(r, a_{n-2}, \dots, a_0)$ as Formula 3.4 and $n - 2$ bound variables $(u_2, \dots, u_{n-1})$, three fewer than before.

Importantly, Formula 3.5 includes a large number of ECs, one fewer than the number of variables. As discussed in Section 3.2.1, the problem should reduce to decomposing a curve. Despite this, the problem was considered too difficult for QEPCAD when $n > 4$, largely due to the variable ordering imposed by the problem.

CAD often performs poorly when applied naively, as many algorithms will fail to detect simplifications or features that can make the problem easier. For this reason, Brown [Bro] suggests the user simplify a problem as much as possible before passing it to a tool such as QEPCAD, particularly when one can eliminate variables. Collins was aware of this, eliminating an additional free variable via substitution, something that an algorithm would miss as free variables are eliminated last. This human-guided early elimination is the inspiration for effective quantification (see Section 3.2.6), a general strategy for eliminating free variables early when possible.

We will examine further simplifications to the Solotareff problem in more detail, first formalising Collins' substitution, then identifying additional simplifications that depend on the value of n . The elimination of an extra variable enables further simplifications by exploiting inequalities among the quantified variables (see Section 3.3.3), reducing the degree of the problem further. If the quantified variables were eliminated first, this additional gain would be lost.

3.3.2 Parity Simplification

Formula (3.5) can be further simplified by using the equation $E_n(-1) = (-1)^{n-1}E_n(+1)$ to solve for one of the coefficients. Upon evaluation of $E_n(x)$ at $x = 1$ and $x = -1$ we obtain the linear *parity equation*

$$\sum_{p \in P} a_p = 1, \quad (3.6)$$

for $P := \{i \mid i \equiv n \pmod{2}\}$. Similarly, we define $Q := \{i \mid i \equiv n + 1 \pmod{2}\}$. We can use (3.6) to obtain a_ν , where $\nu \equiv n \pmod{2}$, in terms of the other a_p , and can substitute the result into Formula (3.5), giving us

$$\begin{aligned} a_\nu &= 1 - \sum_{p \in P \setminus \{\nu\}} a_p \\ (\exists u_2, \dots, u_{n-1}) & \left[(r > S_n) \wedge (-1 < u_2 < \dots < u_{n-1} < 1) \wedge (r > \sum_{q \in Q} a_q) \right. \\ & \left. \wedge_{i=2}^{n-1} [E_n(u_i) = (-1)^{n-i}E_n(1) \wedge E'_n(u_i) = 0] \right], \end{aligned} \quad (3.7)$$

where

$$E_n(x) = x^n + rx^{n-1} - \sum_{\substack{i=0 \\ i \neq \nu}}^{n-2} a_i x^i - \left(1 - \sum_{p \in P \setminus \{\nu\}}^n a_p \right) x^\nu.$$

More specifically,

$$E_n(x) = x^n + rx^{n-1} - \sum_{i=2}^{n-2} a_i x^i - \left(1 - \sum_{\substack{i \text{ odd} \\ i > 1}}^{n-2} a_i \right) x - a_0$$

when n is odd and

$$E_n(x) = x^n + rx^{n-1} - \sum_{i=1}^{n-2} a_i x^i - \left(1 - \sum_{\substack{i \text{ even} \\ i > 0}}^{n-2} a_i \right)$$

when n is even. This reduces the number of equational constraints and the number of variables by 1 each, to $2n - 4$ and $2n - 3$ respectively.

3.3.3 Exploiting Inequalities

In the above formulations of the Solotareff problem, the inequalities ordering the maximal deviation points have attractive properties: they are linear, sparse and symmetric. In addition, this distinction between the points gives a significant amount of information:

$$(-1 = u_1 < u_2 < \dots < u_{n-1} < u_n = 1) \quad \Rightarrow \quad \bigwedge_{\substack{i \neq j \\ 1 \leq i, j \leq n}} u_i - u_j \neq 0.$$

This information is particularly valuable for simplifying the problem. Collins' formulation contains many ECs which are equivalent up to a change of variables. For any function φ , the difference

$$\varphi(u_i) - \varphi(u_j) = \varphi(u_i - u_j) - \varphi(0) = 0$$

will always contain the factor $(u_i - u_j)$. As this factor is known to be non-zero, it can be removed, reducing the degree of the resulting polynomial by one.

When handled algorithmically, these u_i are eliminated first, and the opportunity to exploit this structure is lost. By postponing their elimination, the inequalities are preserved and can be used to reduce degree. Pairing up (i.e. taking the difference of)

ECs of the same form allows this degree drop, while also producing equations symmetric in the two variables, which can be exploited later. If this polynomial is symmetric in u_i and u_j , then it can be expressed in terms of $X_1 = u_i + u_j, X_2 = u_i u_j$, helping to reduce the degree and number of terms further. As such symmetry would be broken when these variables are eliminated, one should usually delay eliminating them as long as possible.

This is another example in which the structure of the input formula can inform the variable ordering: if many polynomial inequalities are present, it may be possible to manipulate the ECs so that these polynomials appear as factors that can be removed, meaning the variables appearing in these inequalities should not be eliminated prematurely.

This observation could also be used to inform variable ordering heuristics. If two variables appear with the same degree, but one appears in many low-degree inequalities, while the other appears in fewer, higher-degree ones, it may be preferable to eliminate the latter, maximising the opportunity to exploit these inequalities. Existing heuristics consider this in a sense: the Brown heuristic [Bro04] (and its EC-aware variant in [Ton21]) will pick variables appearing in fewer terms as a tiebreaker between those with the same overall and total degree, while the `gmods` heuristic [dRE22] chooses the variable with the lowest degree sum, so a variable appearing in many low-degree inequations may be deprioritised.

3.3.4 Further Simplifications

Collins notes the parity equation (3.6) in his paper, but uses (3.5) as the starting point for his workings. However, by continuing from (3.7) and separating out the maximal deviation points u_i into those where i is in P and those where i is in Q , we can continue to eliminate variables and reduce the degree of our equational constraints. We will refer to such points as P -points and Q -points respectively.

Hence, (3.7) becomes:

$$\begin{aligned}
 a_\nu &= 1 - \sum_{p \in P \setminus \{\nu\}} a_p \\
 (\exists u_2, \dots, \exists u_{n-1}) \quad & \left[(r > S_n) \wedge (-1 < u_2 < \dots < u_{n-1} < 1) \wedge (r > \sum_{q \in Q} a_q) \right. \\
 & \wedge [E_n(u_p) - E_n(1) = 0] \\
 & \wedge [E_n(u_q) + E_n(1) = 0] \\
 & \left. \wedge [E'_n(u_i) = 0] \right].
 \end{aligned} \tag{3.8}$$

Each $E_n(u_p) - E_n(1) = 0$ clearly has a factor of $(u_p - 1)$, which is non-zero and can be discarded, reducing the equation to

$$u_p^{n-1} + (1+r)u_p^{n-2} + \sum_{i=0}^{n-3} \left(1+r - \sum_{j>i}^{n-2} a_j\right) u_p^i = 0 \quad (3.9)$$

or

$$\sum_{i=0}^{n-1} \sum_{j>i}^n e_j u_p^i = 0,$$

where e_j is the j th coefficient of $E_n(x)$ (i.e. $e_n = 1, e_{n-1} = r$ and $e_k = -a_k$ for $0 \leq k \leq n-2$).

Substituting in the parity equation allows further factorisation. When n is odd, $a_1 = 1 - \sum_{i \text{ odd}} a_i$ and (3.9) becomes:

$$u_p^{n-1} + (1+r)u_p^{n-2} + \sum_{i=1}^{n-3} \left(1+r - \sum_{j>i}^{n-2} a_j\right) u_p^i + \left(r - \sum_{j>0 \text{ even}}^{n-2} a_j\right) u_p^0 = 0$$

or

$$\sum_{i=1}^{n-1} \sum_{j>i}^n e_j u_p^i + \sum_{j>0 \text{ even}}^n e_j = 0.$$

This has a factor of $u_p + 1$ which can be removed, resulting in

$$\begin{aligned} \sum_{i \text{ odd}}^{n-2} \left(1 - \sum_{j>i \text{ odd}}^{n-2} a_j\right) u_p^i + \sum_{i \text{ even}}^{n-2} \left(r - \sum_{j>i \text{ even}}^{n-2} a_j\right) u_p^i &= 0 \\ \sum_{i \text{ odd}}^{n-2} \sum_{j>i \text{ odd}}^n e_j u_p^i + \sum_{i \text{ even}}^{n-2} \sum_{j>i \text{ even}}^n e_j u_p^i &= 0. \end{aligned} \quad (3.10)$$

When n is even, we have:

$$\begin{aligned} E_n(u_q) + E_n(1) &= u_q^n + r u_q^{n-1} - \sum_{i=1}^{n-2} a_i u_q^i - 1 + r + \sum_{\substack{i \text{ even} \\ i>0}}^{n-2} a_i - \sum_{\substack{i \text{ odd} \\ i>0}}^{n-2} a_i \\ &= u_q^n + r u_q^{n-1} - \sum_{i=1}^{n-2} a_i u_q^i - 1 + r + \sum_{i>0}^{n-2} (-1)^i a_i. \end{aligned}$$

This has a factor of $(u_q + 1)$, which can be discarded, resulting in

$$u_q^{n-1} + (r-1)u_q^{n-2} + \sum_{i=0}^{n-3} \left((-1)^i(1-r) + \sum_{j>i}^{n-2} (-1)^{j-i} a_j \right) u_q^i = 0 \quad (3.11)$$

$$\sum_{i=0}^{n-1} \sum_{j>i}^n e_j (-1)^{j-i+1} u_q^i.$$

For simplicity, we shall refer to the reduced version of $E_n(u_p) - E_n(1)$ and $E_n(u_q) + E_n(1)$ as $\hat{E}_n(u_p)$ and $\hat{E}_n(u_q)$ respectively, i.e:

	n odd	n even
$\hat{E}_n(u_p)$	(3.10) degree: $n-2$	(3.9) degree: $n-1$
$\hat{E}_n(u_q)$	$E_n(u_q) + E_n(1)$ degree: n	(3.11) degree: $n-1$

Table 3.1: The different forms of $\hat{E}_n(u_p)$ and $\hat{E}_n(u_q)$, depending on the parity of n .

In short, (3.8) reduces to:

$$a_\nu = 1 - \sum_{p \in P \setminus \{\nu\}} a_p$$

$$(\exists u_2, \dots, \exists u_{n-1}) \left[(r > S_n) \wedge (-1 < u_2 < \dots < u_{n-1} < 1) \wedge (r > \sum_{q \in Q} a_q) \right. \quad (3.12)$$

$$\left. \wedge [\hat{E}_n(u_p) = 0] \wedge [\hat{E}_n(u_q) = 0] \wedge [E'_n(u_i) = 0] \right].$$

3.3.5 Reconsidering the Variable Ordering

In the formulations of Solotareff's problem given by Collins, one seeks to express the coefficients in terms of r by first eliminating the maximal deviation points $u_2, \dots, u_{n-1}$. The order suggested by the formulation is to first eliminate each quantified u_i , then each a_i until one remains, written in terms of r . This can be repeated with a different ordering to obtain the other free variables, or one can solve and back-substitute.

The internal order in which each u_i and a_i is eliminated may not be significant enough to specify, or can vary. In this case, the variable ordering can be written as

$$r \prec [a_0, \dots, a_{n-2}] \prec [u_2, \dots, u_{n-1}].$$

Variable order can make a huge difference—in this case, we are trying to eliminate higher degree variables before lower degree ones. Note that even in this case, a_ν , a free variable, is eliminated early. In these formulations, we have as many ECs as we have bound and free variables (excluding r , which is to be given).

However, since the a_i are determined by the constraints, we may treat them as existentially quantified. The quantifier-free formula of (3.7) suggests an order on these variables: each a_i appears with degree 1, while each u_i appears with degree n , strongly implying that the a_i should be eliminated first.

In fact, the Brown heuristic (see Definition 2.7.1) suggests a sensible ordering. Stripping (3.7) to just its polynomials (and removing $r > S_n$ as r is not considered for this part of the variable ordering), we have:

- $-1 - u_2$ and $u_{n-1} - 1$, (2 polynomials)
- $r - \sum a_q$, (1 polynomial)
- $u_i - u_{i+1}$, for $2 \leq i \leq n-2$, ($n-3$ polynomials)

and, for $2 \leq i \leq n-1$,

- $u_i^n + ru_i^{n-1} - a_{n-2}u_i^{n-2} + \cdots + a_2u_i^2 + a_{1-\nu}u_i^{1-\nu}$, ($n-2$ polynomials)
- $nu_i^{n-1} + r(n-1)u_i^{n-2} - (n-2)(a_{n-2})u_i^{n-3} + \cdots + 2a_2u_i + (1-\nu)a_{1-\nu}u_i^{-\nu}$. ($n-2$ polynomials)

In this polynomial system:

1. Each u_i appears with max degree n and each a_i with max degree 1, so each a_i should be eliminated before each u_i .
2. Looking at the total degree of the terms in which they appear, each a_i appears with degree $i+1$, and each u_i appears with degree n , so $a_{1-\nu}$ should be eliminated first.
3. Each u_i appears in $2 + (n-\nu) + (n-1)$ terms, $a_{1-\nu}$ appears in $1 + 1 + (1-\nu)$ terms, each other a_q appears 3 times, and each a_p appears twice.

Therefore, the heuristic suggests first eliminating $a_{1-\nu}$. Interestingly, the Tonks heuristic (see Definition 5.2.1), which applies each step of Brown's heuristic first to the ECs, then the inequalities, would instead suggest eliminating $a_{\nu+2}$, as it does not appear in the inequalities, and appears with the lowest total term degree in the ECs. The **gmods** heuristic (see Section 2.7) also suggests eliminating $a_{1-\nu}$ first.

Additionally, by eliminating each a_i first, one preserves the symmetries between each u_i , which can later be exploited to reduce the degree of the remaining expressions.

Using this, we can update the quantifier prefix to

$$a_\nu = 1 - \sum_{p \in P \setminus \{\nu\}} a_p$$

$$(\exists u_2, \dots, u_{n-1}, a_{n-2}, \dots, a_2, a_{1-\nu}) \left[(r > S_n) \wedge (-1 < u_2 < \dots < u_{n-1} < 1) \wedge (r > \sum_{q \in Q} a_q) \right. \\ \left. \wedge [\hat{E}_n(u_p) = 0] \wedge [\hat{E}_n(u_q) = 0] \wedge [E'_n(u_i) = 0] \right].$$

3.3.6 General Elimination Strategies

There is a general method that could be applied to eliminate the other a_i , starting with $a_{1-\nu}$, which depends on the value of n . There are three cases to consider:

Case	n odd ($n \equiv 1 \pmod{4}$)	n even	n odd ($n \equiv 3 \pmod{4}$)
$\hat{E}_n(u_p)$	#: $\frac{n-3}{2}$ (odd no.) degree: $n - 2$	#: $\frac{n-2}{2}$ degree: $n - 1$	#: $\frac{n-3}{2}$ (even no.) degree: $n - 2$
$\hat{E}_n(u_q)$	#: $\frac{n-1}{2}$ (even no.) degree: n	#: $\frac{n-2}{2}$ degree: $n - 1$	#: $\frac{n-1}{2}$ (odd no.) degree: n
$E'_n(u_i)$	#: $n - 2$ degree: $n - 1$	#: $n - 2$ degree: $n - 1$	#: $n - 2$ degree: $n - 1$
Note	Only Q -equations contain a_0 . Use one to eliminate a_0 , then use an equation to eliminate $a_2 \dots$	Every equation contains a_1 . Use one to eliminate a_1 , then another to eliminate $a_2 \dots$	Only Q -equations contain a_0 . Use one to eliminate a_0 , then use an equation to eliminate $a_2 \dots$
To preserve symmetry	Pair up ‘opposite points’ (u_q with u_{n+1-q}) to eliminate a_0 .	Pair up ‘opposite points’ (u_p with u_{n-p} , u_q with u_{n+2-q}) or points and their derivatives to eliminate a_1 .	Use the equations containing the ‘central’ variable $u_{\frac{n+1}{2}}$ to eliminate a_0 and a_2 .
Outcome	$\frac{7n-15}{4}$ ECs $2n - 4$ vars	$n - 2$ ECs $2n - 4$ vars	$2n - 6$ ECs $2n - 5$ vars

Table 3.2: Variable elimination strategies for different values of n .

One could continue to pair up the remaining equations using one of the methods above. Unsurprisingly, eliminating variables this way is infeasible for larger values of n , as doing so results in more variables than equations. However, pairing equations will reduce the degree, because it introduces non-zero factors. Additionally, if this symmetry is

preserved, we can make the substitution $X = u_i + u_j, Y = u_i u_j$ (the elementary symmetric polynomials in u_i and u_j) to further reduce the degree of the polynomials.

The sweet spot here is $n = 5$, as here we have two Q -points, so pairing these up is the same as substituting one into the other, and both a_0 and a_1 have been eliminated using only two ECs. In addition, there is only one ‘central’ P -point, which can be used to eliminate a_2 and a_3 while ensuring the equations are symmetrical around the two Q -points, allowing for the substitution.

3.4 Solutions for $n = 5$

Collins used the Solotareff problem as an example to show the ‘capabilities and limitations of QEPCAD’ [Col96], and noted that the standard formulation of the Solotareff problem when $n = 5$ was too difficult for the version at the time, settling for solving it for when $r = 1$. This could be why he did not look into further simplification. By applying effective quantification and eliminating the a_i first, we were able to simplify the system by hand to something much more tractable. This reordering also produced non-zero factors that could be discarded, further reducing degree and complexity.

We now apply a range of techniques to the $n = 5$ case, using general-purpose techniques such as block ordering, effective quantification and the use of ECs alongside problem-specific strategies which exploit parity, symmetry, and the structure of the inequalities.

We are presented with a partial ordering $r \prec [a, b, c, d] \prec [u, v, w]$, where r is a parameter, a, b, c and d are free variables and u, v and w are bound. The formula contains many low-degree inequalities involving u, v , and w , and many ECs that can be made symmetric in u and w . As the initial problem is written, the aim is to determine $A_5(x)$ by finding its coefficients a, b, c and d in terms of r , but these free variables can be easily eliminated. The equations are linear in these free variables, and c can be solved immediately. Moreover, $A_5(x)$ is known to exist uniquely, so we can choose to eliminate the coefficients first, resulting in equations in u, v, w and r . By solving these in terms of r , we can back-substitute to find a, b, c and d , and therefore $A_5(x)$.

This new elimination order $r \prec [u, v, w] \prec [a, b, c, d]$ preserves the symmetry between u and w , allowing for the use of substitution to reduce degree. The inequality $-1 < u < v < w < 1$ also allows for the removal of non-zero factors, contributing to further simplification.

3.4.1 Applying the Tools

For the case for $n = 5$, we have:

- $A_5(x) = ax^3 + bx^2 + cx + d$,
- $E_5(x) = x^5 + rx^4 - ax^3 - bx^2 - cx - d$

From (3.7), the initial formulation presented in [Col96] gives

$$c = 1 - a$$

$$\begin{aligned}
 (\exists u, v, w, a, b, d) \quad & [(r > S_5) \wedge (-1 < u < v < w < 1) \wedge (r > b + d) \\
 & \wedge [u^5 + ru^4 - au^3 - bu^2 + (a - 1)u + r - 2d - b = 0] \\
 & \wedge [v^5 + rv^4 - av^3 - bv^2 + (a - 1)v - r + b = 0] \\
 & \wedge [w^5 + rw^4 - aw^3 - bw^2 + (a - 1)w + r - 2d - b = 0] \\
 & \wedge [5u^4 + 4ru^3 - 3au^2 - 2bu + a - 1 = 0] \\
 & \wedge [5v^4 + 4rv^3 - 3av^2 - 2bv + a - 1 = 0] \\
 & \wedge [5w^4 + 4rw^3 - 3aw^2 - 2bw + a - 1 = 0]] .
 \end{aligned} \tag{3.13}$$

We must first rewrite $S_5 = 5 \left(\tan^2 \left(\frac{\pi}{10} \right) \right)$ as the root of a polynomial.

Setting $p = \pi/10$, the multiple-angle formula for $\tan(\pi) = \tan(10p) = 0$ gives us

$$t(t^4 - 10t^2 + 5)(5t^4 - 10t^2 + 1) = 0,$$

for $t = \tan(p)$. As t is non-zero, we can discard this first factor, and as $S_5 = 5t^2$, this now becomes

$$(S_5^2 - 50S_5 + 125)(S_5^2 - 10S_5 + 5) = 0,$$

leaving four roots to check. As $0 < S_5 < 5$, we determine that $S_5 = 5 - 2\sqrt{5}$, the root of $x^2 - 10x + 5$ between 0 and 1. With this, we are now able to rewrite $r > S_5$ as $r^2 - 10r + 5 < 0 \vee [r > 1 \wedge r^2 - 10r + 5 \geq 0]$.

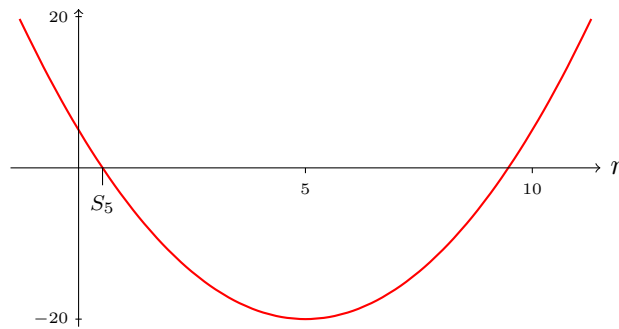


Figure 3-3: Plot of the function $r^2 - 10r + 5$.

With this, and using the improvements from (3.12), the formula becomes:

$$c = 1 - a$$

$$\begin{aligned}
(\exists u, v, w, a, b, d) \quad & [r^2 - 10r + 5 < 0 \vee [r > 1 \wedge r^2 - 10r + 5 \geq 0] \\
& \wedge (-1 < u < v < w < 1) \wedge (r > b + d) \\
& \wedge [u^5 + ru^4 - au^3 - bu^2 + (a - 1)u + r - 2d - b = 0] \\
& \wedge [v^3 + rv^2 + (1 - a)v + r - b = 0] \\
& \wedge [w^5 + rw^4 - aw^3 - bw^2 + (a - 1)w + r - 2d - b = 0] \\
& \wedge [5u^4 + 4ru^3 - 3au^2 - 2bu + a - 1 = 0] \\
& \wedge [5v^4 + 4rv^3 - 3av^2 - 2bv + a - 1 = 0] \\
& \wedge [5w^4 + 4rw^3 - 3aw^2 - 2bw + a - 1 = 0]] .
\end{aligned} \tag{3.14}$$

Only the first and third equations contain the variable d . If we now replace them with their sum and their difference, we obtain an equation that gives a value for d and an equation without it. Their sum gives us

$$d = (u^5 + w^5 + r(u^4 + w^4) - a(u^3 + w^3) - b(u^2 + w^2) - (1 - a)(u + w) - 2b + 2r)/4, \tag{3.15}$$

while taking their difference is equivalent to ‘pairing up’ these equations as in Table 3.2, leaving an equation that is symmetric in u and w . Removing the factor $(u - w)$, this now becomes

$$(u^4 + u^3w + u^2w^2 + uw^3 + w^4) + r(u^3 + u^2w + uw^2 + w^3) - a(u^2 + uw + w^2) - b(u + w) + a - 1 = 0,$$

and the inequality $r > b + d$ becomes

$$2r > u^5 + w^5 + r(u^4 + w^4) - a(u^3 + w^3) - b(u^2 + w^2) - (1 - a)(u + w) + 2b.$$

We have now reduced (3.14) to five equations in the six variables r, a, b, u, v, w :

$$c = 1 - a$$

(3.15)

$$\begin{aligned}
(\exists u, v, w, a, b) \quad & [r^2 - 10r + 5 < 0 \vee [r > 1 \wedge r^2 - 10r + 5 \geq 0] \\
& \wedge (-1 < u < v < w < 1) \\
& \wedge [2r > u^5 + w^5 + r(u^4 + w^4) - a(u^3 + w^3) - b(u^2 + w^2) \\
& \quad - (1 - a)(u + w) + 2b] \\
& \wedge [(u^4 + u^3w + u^2w^2 + uw^3 + w^4) + r(u^3 + u^2w + uw^2 + w^3) \\
& \quad - a(u^2 + uw + w^2) - b(u + w) + a - 1 = 0] \\
& \wedge [v^3 + rv^2 + (1 - a)v - b + r = 0] \\
& \wedge [5u^4 + 4ru^3 - 3au^2 - 2bu + a - 1 = 0] \\
& \wedge [5v^4 + 4rv^3 - 3av^2 - 2bv + a - 1 = 0] \\
& \wedge [5w^4 + 4rw^3 - 3aw^2 - 2bw + a - 1 = 0]] .
\end{aligned} \tag{3.16}$$

We can use the second equation to obtain b in terms of v, a and r and apply this to the other equations and inequalities. The penultimate equation contains the factors $(v - 1)(v + 1)$, which can be removed, giving us:

$$c = 1 - a$$

(3.15)

$$b = v^3 + rv^2 + (1 - a)v + r$$

$$\begin{aligned}
(\exists u, v, w, a) \quad & [r^2 - 10r + 5 < 0 \vee [r > 1 \wedge r^2 - 10r + 5 \geq 0] \\
& \wedge (-1 < u < v < w < 1) \\
& \wedge [(u^2 + w^2 - 2)(v^3 + rv^2 + (1 - a)v) - (u^4 + w^4 - u^2 - w^2)r \\
& \quad - (u^5 + w^5) + a(u^3 + w^3) + (1 - a)(u + w) > 0] \\
& \wedge [(u + w)(v^3 + rv^2 + (1 - a)v) - (u^3 + u^2w + uw^2 + w^3 - u - w)r \\
& \quad - (u^4 + u^3w + u^2w^2 + uw^3 + w^4) + (u^2 + uw + w^2)a + (1 - a) = 0] \\
& \wedge [5u^4 + 4ru^3 - 3au^2 - 2ru - 2uv^3 - 2ruv^2 + 2auv - 2uv + a - 1 = 0] \\
& \wedge [3v^2 + 2rv + 1 - a = 0] \\
& \wedge [5w^4 + 4rw^3 - 3aw^2 - 2rw - 2v^3w - 2rv^2w + 2avw - 2vw + a - 1 = 0]] .
\end{aligned} \tag{3.17}$$

We can now solve the third equation for a in terms of r and v . Substituting this into the remaining equations and inequalities (and again removing non-zero factors), we have:

$$\begin{aligned}
c &= 1 - a \\
(3.15) \\
b &= v^3 + rv^2 + (1 - a)v + r \\
a &= 3v^2 + 2rv + 1
\end{aligned}$$

$$\begin{aligned}
(\exists u, v, w) \quad & [r^2 - 10r + 5 < 0 \vee [r > 1 \wedge r^2 - 10r + 5 \geq 0] \\
& \wedge (-1 < u < v < w < 1) \\
& \wedge [-(u^2 + w^2 - 2)(2v^3 + rv^2) + (u^3 + w^3 - u - w)(3v^2 + 2rv) \\
& \quad - (u^4 + w^4 - u^2 - w^2)r - (u^5 + w^5 - u^3 - w^3) > 0] \\
& \wedge [(2u + 2w)v^3 - (3u^2 + 3uw + 3w^2 - r(u + w) - 3)v^2 \\
& \quad - (2u^2 + 2uw + 2w^2 - 2)rv + (u^3 + u^2w + uw^2 + w^3 - u - w)r \\
& \quad + u^4 + u^3w + u^2w^2 + uw^3 + w^4 - u^2 - uw - w^2 = 0] \\
& \wedge [5u^3 + 5u^2v + 4u^2r - 4uv^2 - 2uvr - 3u - 3v - 2r = 0] \\
& \wedge [5w^3 + 5w^2v + 4w^2r - 4wv^2 - 2wvr - 3w - 3v - 2r = 0]].
\end{aligned} \tag{3.18}$$

We have eliminated four variables, with some significant degree reductions. We have three equations, still one less than the number of variables, in four variables with a maximum degree of 4. If we replace the last two equations with their sum and difference, removing the non-zero factor $(u - w)$ from the latter, these become

$$\begin{aligned}
5(u^3 + w^3) + (u^2 + w^2)(5v + 4r) - (u + w)(4v^2 + 2rv + 3) - 6v - 4r &= 0 \\
5(u^2 + w^2) + (u + w)(5v + 4r) + 5uw - (4v^2 + 2rv + 3) &= 0.
\end{aligned}$$

3.4.2 Exploiting Symmetry

We now have a formula that contains only u , v , w , and r , whose equations and inequalities (aside from $-1 < u < v < w < 1$) are symmetric in u and w , allowing us to rewrite (3.18) in terms of $X = u + w$ and $Y = uw$.

As $-1 < u < v < w < 1$ is not symmetric with respect to u and w , we look to writing it in terms of v , X and Y . If $X = u + w$, $Y = uw$ and $u < w$, then equivalently,

$$u = \frac{X - \sqrt{X^2 - 4Y}}{2}, \quad w = \frac{X + \sqrt{X^2 - 4Y}}{2}.$$

If we consider $S = \sqrt{X^2 - 4Y}$, we have

$$(S < X + 2) \wedge (S > X - 2v) \wedge (S > 2v - X) \wedge (S < 2 - X) \wedge (S^2 = X^2 - 4Y).$$

Squaring each inequality and using the equation gives

$$(Y + X + 1 > 0) \wedge (Y - X + 1 > 0) \wedge (v^2 - vX + Y < 0)$$

with the constraint $(X^2 - 4Y \geq 0)$.

Equation (3.15) gives us a new value for d :

$$(X^5 + rX^4 - (5Y + a)X^3 - (4Yr + b)X^2 + (5Y^2 + 3Ya - (1 - a))X + 2((Y^2 + 1)r + (Y - 1)b))/4, \quad (3.19)$$

and our formula becomes

$$\begin{aligned} c &= 1 - a \\ d &= (3.19) \\ b &= v^3 + rv^2 + (1 - a)v + r \\ a &= 3v^2 + 2rv + 1 \\ (\exists X, Y, v) \quad & [r^2 - 10r + 5 < 0 \vee [r > 1 \wedge r^2 - 10r + 5 \geq 0] \\ & \wedge (Y + X + 1 > 0) \wedge (Y - X + 1 > 0) \wedge (v^2 - Xv + Y < 0) \\ & \wedge [2Xv^3 - (3X^2 - rX - 3Y - 3)v^2 - 2(X^2 - Y - 1)rv \\ & \quad + (X^3 - (2Y + 1)X)r + X^4 - (3Y + 1)X^2 + Y^2 + Y = 0] \\ & \wedge [5(X^3 - 3XY) + (X^2 - 2Y)(5v + 4r) - (4v^2 + 2rv + 3)X \\ & \quad - 6v - 4r = 0] \\ & \wedge [5(X^2 - Y) + (5v + 4r)X - 4v^2 - 2rv - 3 = 0]] . \end{aligned} \quad (3.20)$$

This last equation gives a value for v^2 as a polynomial which is linear in v , which can be used to remove higher powers (see Section 3.2.2):

$$v^2 = ((5X - 2r)v + 5(X^2 - Y) + 4Xr - 3)/4. \quad (3.21)$$

Using this, the other equations become

$$\begin{aligned} & (15X^3 + 2rX^2 + 10XY + 4Yr + 18X + 4r)v \\ & + 4(X^3 + 2XY + 4X)r + 3X^4 + (11Y + 25)X^2 - 22Y^2 - 40Y - 18 = 0, \\ & (5Y + 3)v + 5XY + 4Yr + 2r = 0, \end{aligned} \quad (3.22)$$

the latter of which can give a value for v as long as its leading coefficient is not zero.

From here, these equations can be solved by elementary means. We consider some special cases below.

Let us first consider when $Y = -3/5$. The final equation gives us $X = -2r/15$. Substituting these into (3.20) or (3.21) gives $v = -r/3$. Using these, we obtain the equation $3r^4 + 410r^2 + 675 = 0$, which has no real solutions. This tells us that $Y \neq -3/5$ and we have a value for v . Using this value, we have the following equations:

$$(8X^2Y + 4X^2 + 16Y^2 + 24Y + 8)r^2 + (50X^3Y + 18X^3 + 20XY^2 + 8XY - 12X)r + 60X^4Y - 9X^4 - 5X^2Y^2 - 68X^2Y + 110Y^3 - 75X^2 + 266Y^2 + 210Y + 54 = 0$$

$$(24Y^2 + 20Y + 4)r^2 + (110Y^2 + 40Y - 6)Xr + (100Y^2 - 75Y - 45)X^2 + 125Y^3 + 225Y^2 + 135Y + 27 = 0$$

These have degree 5 and 4 respectively. Note that the latter is quadratic in X :

$$(60Y - 9)X^4 + (50Y + 18)rX^3 + ((8Y + 4)r^2 - 5Y^2 - 68Y - 75)X^2 + (20Y^2 + 8Y - 12)rX + (16Y^2 + 24Y + 8)r^2 + 110Y^3 + 266Y^2 + 210Y + 54 = 0,$$

$$(100Y^2 - 75Y - 45)X^2 + (110Y^2 + 40Y - 6)rX + (24Y^2 + 20Y + 4)r^2 + 125Y^3 + 225Y^2 + 135Y + 27 = 0.$$

$100Y^2 - 75Y - 45 = 0$ has two solutions, but only one fits $-1 < Y < 1$, namely

$$Y = \frac{3}{8} - \frac{3\sqrt{105}}{40} \approx -0.3935213078.$$

Substituting this in to our two equations, the latter gives

$$X = -\frac{(5\sqrt{105} - 63)(-448r^2 + 405\sqrt{105} - 945)}{161280r},$$

but we know $r > 0$. If we wanted to substitute this into the former, we obtain values for r , one of which is a positive real number, namely

$$r = \frac{\sqrt{-164135790 + 3862950\sqrt{105} + 2730\sqrt{93744545370 - 8730591618\sqrt{105}}}}{9464}.$$

We can continue, substituting these into (3.20) to get

$$v = -\frac{40r^2\sqrt{105} - 504r^2 - 189\sqrt{105} + 2025}{72r(\sqrt{105} - 13)}.$$

From this, we obtain precise values for the coefficients a , b , c , and d , and have following numerical approximations:

$$\begin{aligned} r &\approx 0.7772073152, & X &\approx 0.2754247409, & Y &\approx -0.3935213078 \\ u &\approx -0.5045383563, & v &\approx 0.2042859171, & w &\approx 0.7799630971 \\ a &\approx 1.442743234, & b &\approx 0.7277215058, & c &\approx -0.4427432339, & d &\approx -0.0780878820 \end{aligned}$$

If we now consider when $(100Y^2 - 75Y - 45)$ is non-zero, we can eliminate this again, substituting the second into the first to produce an equation containing $(5Y + 3)$ in the numerator and $(20Y^2 - 15Y - 9)$ in the denominator. These are both non-zero, so can be removed, resulting in the product of two sextics:

$$\begin{aligned} &(275625Y^6 - (23200r^2 + 78750)Y^5 + (576r^4 - 17800r^2 - 41625)Y^4 \\ &+ (960r^4 - 2080r^2 + 35100)Y^3 + (592r^4 + 816r^2 - 2025)Y^2 \\ &+ (160r^4 + 576r^2 - 2430)Y + 16r^4 + 216r^2 + 729) \cdot \\ &(2500Y^6 + (200r^2 + 10000)Y^5 - (16r^4 - 300r^2 - 16000)Y^4 \\ &- (80r^4 + 420r^2 - 12900)Y^3 - (144r^4 - 1116r^2 - 5400)Y^2 \\ &- (108r^4 + 756r^2 - 1080)Y - 27r^4 - 162r^2 + 81) = 0. \end{aligned} \tag{3.23}$$

We can also obtain this by taking the X -resultant. Alternatively, one can compute the Y -resultant to obtain another product of two sextics.

These roots depend on r and so which roots are valid depends on the value of r . We do not attempt to find a closed form solution for r , but this allows us to compute a solution for any specified value of r without needing to invoke CAD at all. For example, when $r = 1$, the above product becomes

$$\begin{aligned} &4(275625Y^6 - 101950Y^5 - 58849Y^4 + 33980Y^3 - 617Y^2 - 1694Y + 961) \cdot \\ &(625Y^4 + 1300Y^3 + 846Y^2 + 108Y - 27)(Y + 1)^2 = 0. \end{aligned}$$

The first factor has no real roots, and the second has two: α_1 between -0.419919014 and -0.4199185371 , and α_2 between 0.1185973 and 0.1185976 . Taking $Y = \alpha_2$ gives $X \approx -1.002978726$, but no value of v satisfies the system (3.20). However, taking $Y = \alpha_1$ gives $X \approx 0.2298934353$, and thus $u \approx -0.5431806829$ and $w \approx 0.7730741182$. Continuing to back-substitute returns the same values given by Collins.

Chebyshev's theorem guarantees that the solution is unique, and note that in these calculations the ECs show there are finitely many solutions (i.e. we have shown the solution set is a curve, as promised in Section 3.2.1), and from there the inequalities can be used to show the solution is unique.

Chapter 4

CAD in *Macaulay2*

4.1 Introduction

This chapter documents and presents the `CylindricalAlgebraicDecomposition` package [Lee25b] for *Macaulay2* [GS], which enables users to solve satisfiability problems in non-linear real arithmetic using open cylindrical algebraic decomposition (open CAD).

CAD is well suited for computation and has been implemented in many widely used computer algebra systems such as *REDUCE*, *Maple*, and *Mathematica*. However, this package provides the first implementation of CAD within *Macaulay2*. The work presented in this chapter is also described in the accompanying preprint [LdR25]¹.

The package reduces the complexity of the original algorithm proposed by Collins by incorporating modern improvements. It implements open CAD (Section 2.8.4), uses the contemporary Lazard projection (Section 2.6.1), and is the first implementation of the `gmods` heuristic for variable ordering (Section 2.7). In addition to this package are some minor fixes and improvements to the `RealRoots` package’s method of real root isolation, upon which this algorithm builds.

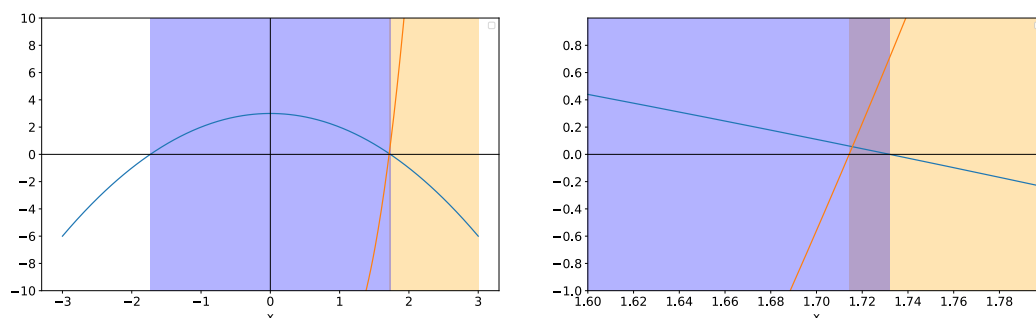
¹Also under review by the *Journal of Software for Algebra and Geometry*.

4.1.1 Motivating Example

Imagine we are interested in determining whether there exists an $x \in \mathbb{R}$ such that

$$g(x) = 3 - x^2 > 0 \quad \text{and} \quad h(x) = (7x - 12)(x^2 + x + 1) > 0. \quad (4.1)$$

Studying the problem numerically, one could sample 1001 equispaced points between -50 and 50 and find no point that satisfies the conditions. However, such a point does exist:



- (a) It is not initially clear whether there is some value of x that ensures both polynomials are positive... (b) ...however it becomes clear that such an x does exist when looking in greater detail.

Figure 4-1: Algebraic varieties of $3 - x^2$ (blue) and $(7x - 12)(x^2 + x + 1)$ (orange), and the associated values of x ensuring these polynomials are positive (blue and orange highlighting, respectively).

The function `findPositiveSolution` in this package can check whether a set of strict polynomial inequalities has a solution. For the univariate example above, this function outputs the following:

```
i1 : R=QQ[x]

i2 : findPositiveSolution({3-x^2,(7*x-12)*(x^2+x+1)})

1539
o2 = (true, HashTable{x => ----})
896
```

The package is not restricted to the univariate case and can find solutions for polynomials in as many variables as desired, given enough time.

4.2 Design Choices for the package

Certain decisions have been made in designing the CAD algorithm to best suit this package. These decisions are explained below.

`CylindricalAlgebraicDecomposition` employs `gmods`, a strategy developed by del Río and England [dRE22], which first projects the variable with the lowest degree sum across the set of polynomials (see Section 2.7).

The chosen projection operator is the *Lazard projection* (see Section 2.6.1); constructing an open CAD eliminates the need for extra checks that are needed when applying the Lazard projection in a full CAD.

In `CylindricalAlgebraicDecomposition`, the set of input and output polynomials are replaced by their non-constant irreducible factors, ensuring they are irreducible and pairwise coprime.

4.2.1 Open CAD

Recall from Section 2.8.4 the notion of an *open CAD*, that is, open solution sets of systems of strict polynomial inequalities [Str00]. In other words, an open CAD will only consist of open, full-dimensional n -cells.

At present, *Macaulay2* does not support polynomial division over the reals or symbolic root storage, making a full CAD impossible due to the inability to store or lift over algebraic irrational roots. Despite this limitation, many real-world applications are not interested in these zero-measure regions, so an open CAD is enough for such cases.

We have implemented an open CAD algorithm, using the existing `RealRoots` package [GMSY24] to isolate roots within rational intervals, allowing rational sample points to be chosen which represent every open cell. Additionally, issues and inefficiencies within the `RealRoots` package have been corrected and improved.

4.2.2 Improvements to RealRoots

`CylindricalAlgebraicDecomposition` calls the existing `RealRoots` package to perform real root isolation. However, there are some limitations and inefficiencies in `RealRoots` that we have addressed:

- **Only works on one polynomial:** `CylindricalAlgebraicDecomposition` simply performs this real root isolation on $\prod f_i$, where each f_i is reduced to its irreducible factors during the projection phase.

- **Inefficient root bound:** The original root bound was weaker than necessary, and would scale badly in cases of coefficient blowup. The fix improves the existing bound, implements another which better handles large coefficients, and the command now takes the smaller bound, speeding up each real root isolation step.
- **Inefficient with close roots:** The original algorithm would continue to refine every interval in cases when one root required a smaller interval, creating extra work. Now, once a root has been sufficiently isolated, its interval is left alone.
- **Roots not always ordered:** Root intervals are now consistently listed lowest to highest.

4.3 The CylindricalAlgebraicDecomposition Package

This section provides a detailed description of `CylindricalAlgebraicDecomposition`. This package allows the user to produce an open, sign-invariant CAD with respect to a list of input polynomials in $\mathbb{Q}[x_1, \dots, x_n]$ by constructing a tree of sample points in $\mathbb{Q}^n$, representing the open cells of a CAD of $\mathbb{R}^n$. It also allows users to step through each phase of the open CAD algorithm and check for points where all input polynomials are positive.

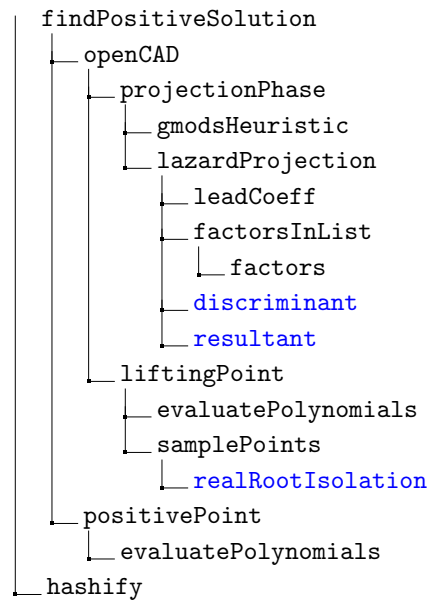


Figure 4-2: List of functions contained in the `CylindricalAlgebraicDecomposition` package. Methods labelled in blue are from external packages.

4.3.1 Overview

The package's top-level function is `findPositiveSolution`, which takes a list of input polynomials and returns a point in which all input polynomials are strictly positive, or informs the user that no such point exists.

This is built upon two other functions: `openCAD`, which constructs an open CAD for the given list of polynomials as a tree of sample points, and `positivePoint` which takes the tree (or a branch with leaves) and checks if such a positive point exists. These commands are also available to the user, allowing them to construct open CADs and check the cells as needed.

We now proceed to describe these functions in more detail before continuing with the remaining functions in the package, providing inputs, outputs, and examples. To illustrate the algorithm, we will use the following system of polynomials $\mathcal{F} = \{f_1, f_2\}$, taken from [Jir95]:

$$\begin{aligned} f_1 &= x_1^2 + x_2^2 - 1 \\ f_2 &= x_1^3 - x_2^2. \end{aligned} \tag{4.2}$$

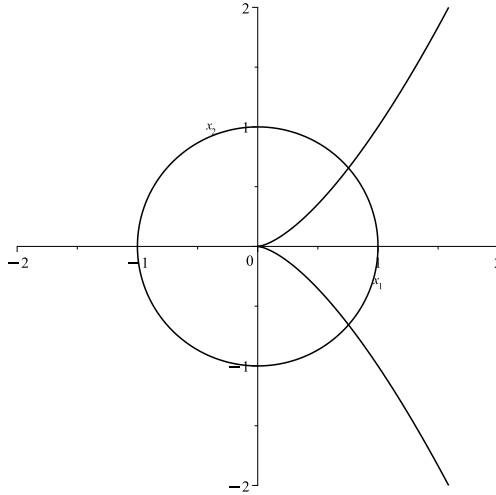


Figure 4-3: The system of polynomials $\mathcal{F} = \{f_1, f_2\}$, used as an example throughout this section.

```
R=QQ[x1,x2];
f1:=x1^2+x2^2-1; f2:=x1^3-x2^2;
F={f1,f2};
```

openCAD

Input: List of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in variables $x_1, \dots, x_n$.

Output: Tree of sample points representing the open CAD of $\mathbb{R}^n$ with respect to $\mathcal{F}_n$.

The command `openCAD` produces an open, projection and lifting-style CAD, and consists of a projection phase (see Section 4.3.2) and a lifting phase (see Section 4.3.3).

The format of `openCAD` is a tree, represented in *Macaulay2* as a collection of nested hash tables with n levels. Each level k , for $1 \leq k < n$, represents the CAD of $\mathbb{R}^k$, containing the projection polynomials $\mathcal{F}_k$, the cell below (as a $(k - 1)$ -dimensional sample point) and the collection of sample points in x_k obtained by lifting over that cell. Each leaf node (level n) represents an n -dimensional sample point of $\mathcal{F}_n$.

The output of `openCAD` is substantial, and therefore is left unprinted in order to minimise clutter (see `hashify` below for the full output).

```
i3 : openCAD(F)
o3 = MutableHashTable{...7...}
```

positivePoint

Input 1: List of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in variables $x_1, \dots, x_n$.

Input 2: A tree of sample points representing an open CAD (or a subtree reaching down to level n).

Output: If every input polynomial is positive at an n -dimensional sample point, that sample point is returned. If not, `null` is returned.

```
i5 : positivePoint(F,openCAD(F))
o5 = MutableHashTable{...2...}
```

Yes, such a point exists. The point is:

```
i6 : peek positivePoint(F,openCAD(F))
o6 = MutableHashTable{x2 => 0 }
      17
      x1 => --
      8
```

findPositiveSolution

Input: List of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in variables $x_1, \dots, x_n$.

Output: Returns a boolean indicating if there exists a point in which all polynomials are strictly positive, and if so, it returns one such point as a hash table.

The command **findPositiveSolution** combines the previous commands, first producing an open CAD from the input polynomials, then checking this for a point where the input polynomials are positive.

```
i7 : findPositiveSolution(F)
                                     17
o7 = (true, HashTable{x1 => --})
                                     8
                                     x2 => 0
```

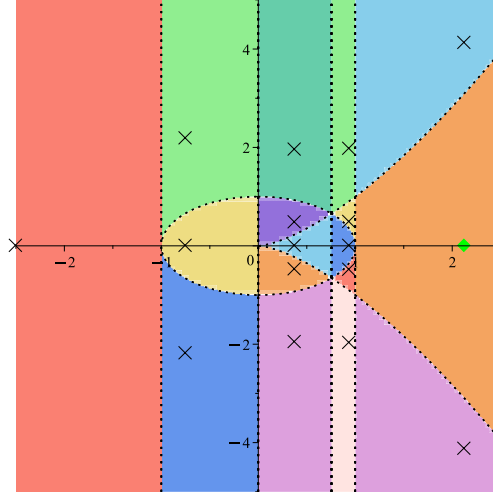


Figure 4-4: A visualisation of the open CAD of $\mathcal{F}$. The coloured regions represent the open cells, the green diamond the sample point from **findPositiveSolution**, and the black crosses the remaining sample points from **openCAD**.

Figure generated by **GraphCAD** (see Chapter 6).

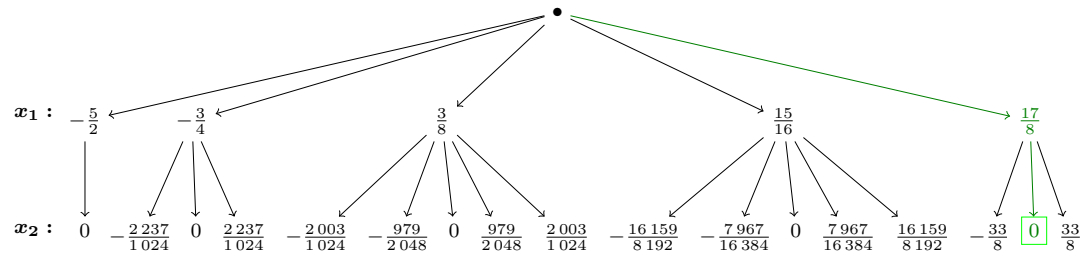


Figure 4-5: A representation of these sample points in a tree structure.

`hashify`

Input: A data structure.

Output: The same data structure with any mutable hash tables converted to immutable ones.

The recursive nature of this open CAD algorithm greatly benefits from the ability to copy and modify previous steps, something only possible with a *mutable* hash table. Additionally, mutable hash tables do not display their entries by default, avoiding the otherwise overwhelming output given by `openCAD` and allowing the user to expand the parts of interest as they see fit.

However, mutable hash tables are incomparable in *Macaulay2* and, additionally, cannot be fully expanded. The user is able to see the next level using `peek`, but this does not reveal anything further. In this example, the first level consists of $\mathcal{F}_1$ (`polynomials`), the (empty) cell below (`point`), and the five open sample points of the CAD of $\mathbb{R}^1$:

```
i8 : peek(openCAD(F))

o8 = MutableHashTable{point => MutableHashTable{
                                     3      2
      polynomials => {x1 - 1, x1 + 1, x1, x1 + x1 - 1}
      3
      - - => MutableHashTable{...5...}
      4
      5
      - - => MutableHashTable{...2...}
      2
      3
      - => MutableHashTable{...7...}
      8
      15
      -- => MutableHashTable{...7...}
      16
      17
      -- => MutableHashTable{...5...}
      8
    }
```

The `hashify` command was implemented to address this, converting mutable hash tables into immutable ones which are viewable in their entirety. Fully expanding the output of `openCAD` illustrates the structure of the computed open CAD (compare this with Figure 4-5), while also demonstrating that `hashify` should be used with care:

i9 : hashify openCAD(F)

o9 =

```

      3      2237      3
HashTable{- - => HashTable{- ---- => HashTable{point => HashTable{x1 => - - }} }
      4      1024      4
                                2237
                                x2 => - ----
                                1024
                                3
                                4
0 => HashTable{point => HashTable{x1 => - -}}
                                4
                                x2 => 0
                                2237      3
---- => HashTable{point => HashTable{x1 => - - }}
1024      4
                                2237
                                x2 => ----
                                1024
                                3
point => HashTable{x1 => - -}
                                4
                                2 7 2 27
polynomials => {x2 - --, - x2 - --}
                                16 64
5
- - => HashTable{0 => HashTable{point => HashTable{x1 => - -}}}
2      2
                                x2 => 0
                                5
point => HashTable{x1 => - -}
                                2
                                2 21 2 125
polynomials => {x2 + --, - x2 - --}
                                4 8
3      979      3
- - => HashTable{- ---- => HashTable{point => HashTable{x1 => - }} }
8      2048      8
                                979
                                x2 => - ----
                                2048
                                2003      3
- ---- => HashTable{point => HashTable{x1 => - }}
1024      8
                                2003
                                x2 => - ----
                                1024
                                3
0 => HashTable{point => HashTable{x1 => -}}
                                8
                                x2 => 0
                                979      3
---- => HashTable{point => HashTable{x1 => - }}
2048      8
                                979
                                x2 => ----
                                2048
2003      3
---- => HashTable{point => HashTable{x1 => - }}
1024      8
                                2003
                                x2 => ----
                                1024
                                3
point => HashTable{x1 => -}
                                8
                                2 55 2 27
polynomials => {x2 - --, - x2 + ---}
                                64 512
15      7967      15
-- => HashTable{- ---- => HashTable{point => HashTable{x1 => - }} }
16      16384      16
                                7967
                                x2 => - ----
                                16384
                                16159      15
- ---- => HashTable{point => HashTable{x1 => - }}
8192      16
                                16159
                                x2 => - ----
                                8192
                                15
0 => HashTable{point => HashTable{x1 => -}}
                                16
                                x2 => 0
                                7967      15
---- => HashTable{point => HashTable{x1 => - }}
16384      16
                                7967
                                x2 => ----
                                16384
                                16159      15
---- => HashTable{point => HashTable{x1 => - }}
8192      16
                                16159
                                x2 => ----
                                8192
                                15
point => HashTable{x1 => -}
                                16
                                2 31 2 3375
polynomials => {x2 - --, - x2 + ----}
                                256 4096
17      33      17
-- => HashTable{- -- => HashTable{point => HashTable{x1 => - }} }
8      8
                                33
                                x2 => - --
                                8
                                17
0 => HashTable{point => HashTable{x1 => -}}
                                8
                                x2 => 0
                                33      17
-- => HashTable{point => HashTable{x1 => -}}
8      8
                                33
                                x2 => --
                                8
                                17
point => HashTable{x1 => -}
                                8
                                2 225 2 4913
polynomials => {x2 + ---, - x2 + ----}
                                64 512
point => HashTable{}
                                3 2
polynomials => {x1 - 1, x1 + 1, x1, x1 + x1 - 1}

```

We are now able to see the full structure of **openCAD**: each level of this tree corresponds to a variable, listing sample points based on those of the level below. In this example, we can see the full list of open sample points of the CAD of $\mathcal{F}$. For a deeper explanation of this structure, see Section 4.3.3.

4.3.2 Projection Phase

The entire projection process is handled by the command `projectionPhase`, which takes the list of input polynomials $\mathcal{F}_n$ and returns the full collection of projection polynomial lists $\mathcal{P} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$, along with the corresponding variable ordering $x_1 \prec \dots \prec x_n$. At each step, `gmodsHeuristic` selects a variable to project, and `lazardProjection` constructs the resulting projection polynomials.

`gmodsHeuristic`

Input 1: A list of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in n variables.

Input 2: The list of variables $x_1, \dots, x_n$ appearing in $\mathcal{F}_n$.

Output: The variable to project as suggested by the `gmods` heuristic (see Section 2.7).

```
i10 : gmodsHeuristic(F,support(F))
o10 = x2
```

`lazardProjection`

Input 1: List of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in variables $x_1, \dots, x_n$.

Input 2: The variable to project away, typically the greatest with respect to $\prec$, x_n .

Output: A list of projection polynomials $\mathcal{F}_{n-1}$ in $x_1, \dots, x_{n-1}$, obtained via the Lazard projection with respect to x_n .

This command ensures polynomials are irreducible and pairwise coprime using the subcommand `factorsInList` to break them into their unique irreducible factors, discarding constants. The Lazard projection is then applied with respect to x_n , returning the list of factors of the leading and trailing coefficients, discriminants and resultants.

Projecting $\mathcal{F}$ with respect to x_2 returns:

```
i11 : lazarProjection(F,x2)
o11 = {x1^3 - 1, x1^2 + 1, x1^3 + x1^2 - 1}
```

where

- $x_1 - 1$ and $x_1 + 1$ are factors of the trailing coefficient and discriminant of f_1 ,
- x_1 is a factor of the trailing coefficient of f_2 , and
- $x_1^3 + x_1^2 - 1$ is the resultant of f_1 and f_2 .

`projectionPhase`

Input: List of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in variables $x_1, \dots, x_n$.

Output 1: List of projection polynomial lists $\mathcal{P} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$ (see Section 2.5 for more details).

Output 2: The associated variable ordering.

```
i12 : projectionPhase(F)
      3      2      2      2      3      2
o12 = ({x1 - 1, x1 + 1, x1, x1 + x1 - 1}, {x1 + x2 - 1, x1 - x2 },
      {x1,x2})
```

4.3.3 Lifting Phase

As with the projection phase, the full lifting phase is handled by one command. Given a list of projection polynomial lists $\mathcal{P}$, their associated ordering, and a $(k-1)$ -dimensional sample point representing the cell below, the command `liftingPoint` recursively constructs cells above each sample point until a full collection of n -dimensional sample points is obtained.

Ordinarily (as is the case for `openCAD`, see Section 4.3.1), this begins with an empty (0-dimensional) sample point representing the root node of the tree, resulting in the construction a CAD of $\mathbb{R}^n$. This command also allows the construction of a ‘sub-CAD’ of the space above any provided sample point.

Each sample point is stored as a hash table mapping variables to values. The command `liftingPoint` calls `evaluatePolynomials` to evaluate polynomials above the sample point, returning a list of univariate polynomials. This is then passed to `samplePoints`, which isolates roots and returns the open sample points of the space.

`samplePoints`

Input: List of univariate polynomials $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[x]$.

Output: A list of open sample points for the CAD of the (one-dimensional) space containing $\mathcal{F}$.

This command applies `realRootIsolation` from `RealRoots` to the polynomials, isolating each real root within a rational interval, then returns sample points either side of these intervals. If applied to our projection polynomials $\mathcal{F}_1$, these are the open sample

points of $\text{CAD}(\mathcal{F}_1)$, but if applied to $\mathcal{F}_k$ evaluated at a $(k-1)$ -dimensional sample point, these are the open sample points of the CAD of the space above that point.

For example, we can obtain the open sample points of our projection polynomials obtained in Section 4.3.2:

```
i13 : samplePoints(lazardProjection(F,x2))
```

```

      5      3  3  15  17
o13 = {- -, - -, -, --, --}
      2      4  8  16  8

```

These are the values of x_1 seen in Figure 4-5.

evaluatePolynomials

Input 1: List of polynomials $\mathcal{F}_n = \{f_1, \dots, f_r\}$ in variables $x_1, \dots, x_n$.

Input 2: A k -dimensional point.

Output: The list of polynomials evaluated at this point.

Above one of these sample points, our new univariate polynomials become:

```
alpha = new MutableHashTable; alpha#x1 = -3/4;
```

```
i14 : evaluatePolynomials(F,alpha)
```

```

      2      7      2  27
o14 = {x2  - --, - x2  - --}
      16      64

```

We can then look at the open sample points of x_2 above this:

```
i15 : samplePoints(evaluatePolynomials(F,alpha))
```

```

      2237      2237
o15 = {- ----, 0, ----}
      1024      1024

```

liftingPoint

Input 1: List of projection polynomial lists $\mathcal{P} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$.

Input 2: The associated variable ordering $x_1 \prec \dots \prec x_n$.

Input 3: A $(k - 1)$ -dimensional point.

Output: An open CAD (see Section 2.5) over the given point.

Above each sample point of the previous level, a cell is constructed. This level k cell contains:

- the $(k - 1)$ dimensional sample point below (point),
- the polynomials of $\mathcal{F}_k$ evaluated at this sample point (polynomials),
- the values of x_k for each k -dimensional sample point above point. The command `liftingPoint` is called again with these new sample points.

At the top level (level n), each cell only contains point, the sample point for that open cell.

Lifting above the previous sample point:

```
(PP,ord) = projectionPhase(F);

i16 : hashify(liftingPoint(PP,ord,alpha))

          2237
o16 = HashTable{- ---- => HashTable{point => HashTable{x1 => - -   }}}
          1024                      4
                                   2237
                                   x2 => - ----
                                   1024

                                   3
0 => HashTable{point => HashTable{x1 => - -}}
                                   4
                                   x2 => 0

          2237
----- => HashTable{point => HashTable{x1 => - - }}
          1024                      4
                                   2237
                                   x2 => ----
                                   1024

                                   3
point => HashTable{x1 => - -}
                                   4

          2    7    2    27
polynomials => {x2 - --, - x2 - --}
                  16    64
```

This demonstrates the lifting phase: we first isolate their real roots of the univariate polynomials in $\mathcal{F}_1$ and select sample points from each open cell around the roots. These open sample points are then substituted into $\mathcal{F}_2$ to produce a new list of univariate polynomials, and the process repeats until a full open CAD is constructed.

4.4 Performance of CAD on Intersecting Hyperspheres

It is known that CAD has doubly-exponential complexity, meaning that only problems in fewer variables can be solved. To measure the performance of our package, we created CADs for the following intersecting n -dimensional spheres (of which the three-dimensional case is depicted in Figure 4-6):

$$\mathcal{F}_n = \left\{ \sum_{i=1}^n (x_i - 1)^2 - 4, \sum_{i=1}^n (x_i + 1)^2 - 4 \right\}$$

The runtimes achieved by the package are the following:

Variables	Runtime (seconds)	n -cells
1	0.0308762	5
2	0.335376	29
3	6.06833	467
4	1883.13	7370

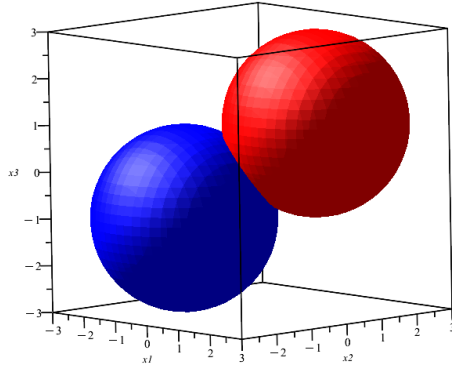


Figure 4-6: A plot of the algebraic variety of $\mathcal{F}_n$.
Two intersecting spheres centered at $(1,1,1)$ and $(-1,-1,-1)$ respectively.

Chapter 5

Benchmarking CAD Algorithms in *Maple*

5.1 Introduction and Purpose

The goal of this chapter is to compare two software implementations of CAD (PLCAD and RCCAD), implementing two different approaches (PL-CAD and RC-CAD), but both based on the same underlying computer algebra system. We provide an initial, exploratory benchmarking comparing their performance, with the aim to help identify the strengths and weaknesses of each method, and highlight features and problem types in which one is more effective. This will inform future research and implementations, including the potential for a meta-algorithm which suggests one algorithm over the other when presented with a given formula as input. The two methods we will compare are:

- PLCAD, a method based on projection and lifting (PL-CAD), implemented in the `QuantifierElimination` package, and
- RCCAD, a method based on regular chains (RC-CAD), implemented in the `RegularChains` package.

These represent modern, state-of-the-art implementations of their respective theories. PLCAD builds upon several decades of evolution in projection and lifting methods, while RCCAD employs comparatively new techniques. Experimentally, RCCAD has been shown to be very promising.

Aside from a comparison in [Ton21], which introduces the `QuantifierElimination` package, no other benchmarking or comparison between these two implementations seems to exist, despite their availability. Prior results [CMM14a, Wil14] suggest that RCCAD can outperform earlier PL-CAD implementations, with [Ton21] suggesting that it is competitive with PLCAD. Despite this promise, the RC-CAD approach has received far less attention.

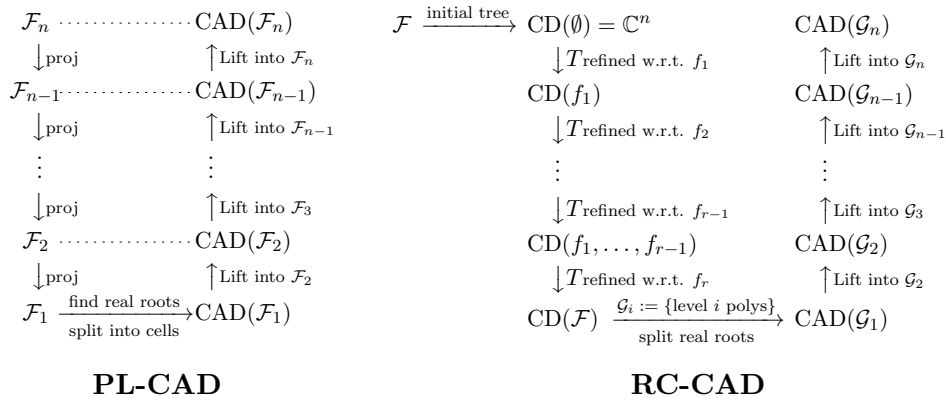


Figure 5-1: An overview of each method. Note that PL-CAD counts down dimensions, whereas RC-CAD counts down polynomials.

5.2 Background and Implementation

For the purposes of benchmarking, we focus on the commands used to produce a CAD in *Maple*, both of which are called `CylindricalAlgebraicDecompose`. The underlying algorithms are already well-documented in the literature (see Section 2.5.1 and Section 2.9), so only aspects relevant to these specific algorithms will be discussed here. For an in-depth look at each package, see [CMM14a, CMM14b, CMM14c, Ton21, Mapa, Mapb].

Both algorithms use modern advances and include various features which make them suitable for comparison. PLCAD is the first in *Maple* to use the Lazard projection operator, with full support for equational constraints (ECs) and preprocessing using Gröbner bases. It accepts Tarski formulae as input and offers multiple variable ordering heuristics, the default being an EC-aware modification of the Brown heuristic known as the Tonks heuristic.

RCCAD uses the incremental algorithm described in [CMM14a, Wil14] and Section 2.9.4. Instead of using a projection operator, this algorithm incrementally refines a decomposition with respect to the input polynomials, often resulting in fewer cells. This package

also supports ECs and even allows for the production of a TTICAD, and has its own variable ordering based on the Brown heuristic [Ton21]. However, it expects input in the form of a list of lists, and this distinction and the associated challenges are detailed later in this chapter.

PLCAD and QuantifierElimination

The `QuantifierElimination` package provides a state-of-the-art PL-CAD implementation. The package has a strong focus on ECs, is able to apply restricted and semi-restricted Lazard projections throughout, and can handle certain types of curtains when using these restricted projections.

For an input $\mathcal{F}$, which may be a list of polynomials or an unquantified Tarski formula:

```
PLCAD:= QuantifierElimination:-CylindricalAlgebraicDecompose(F,options),
```

whose relevant options are:

- `variablestrategy`: `tonks` (default), or a supplied list of variables. Other options are `brown`, `ndrr`, `sotd`, `greedy`.
- `propagateecs`: `true` (default), `false`.
- `useequations`: `multiple` (default), `single`, `none`.
- `usegroebner`: `true`, `false`.

This command additionally accepts separate lists of polynomial inequalities and ECs as input, and any ECs detected in the input are used for restricted projection and variable ordering. Internally, it operates on its own data types, allowing close integration with other commands in the `QuantifierElimination` package, which can export data in formats usable by many pre-existing functions. A key feature of this implementation is its use of Lazard-style PL-CAD, which can handle curtains, unlike the earlier McCallum projection. Various heuristics are available, but neither the variable ordering chosen nor the projection polynomials produced are directly accessible by the user, being used internally.

RCCAD and RegularChains

The `RegularChains` package uses the incremental RC-CAD algorithm as default (with support for the earlier recursive algorithm of [CMMXY09]), and also allows for the handling of ECs and supports TTICAD. Unlike PL-CAD, this will never have issues

with curtains. There are many algorithmic decisions made to avoid unnecessary computation and improve performance [CMM14a]. It takes as input a list of polynomials or a ‘list of semi-algebraic systems’ representing a DNF-like disjunction of conjunctions of polynomial constraints, and requires the user to specify a polynomial ring in advance, as is the case in *Macaulay2* (see Chapter 4).

For such an input $\mathcal{F}$:

```
RCCAD:= RegularChains:-SemiAlgebraicSetTools:-
CylindricalAlgebraicDecomposition(F, R, outopt, methopt, optopt),
```

where:

- `R:=RegularChains:-PolynomialRing(V)` for `V` either a supplied list of variables or `RegularChains:-SuggestVariableOrder(F,decomposition = cad)`.
- `outopt`: Output options: `allcell/cadcell`. Other options include `rootof`, `tree`, `list` and `piecewise`.
- `methopt`: Method options: `incremental` [CMM14a] (default), or `recursive` [CMMXY09].
- `optopt`: Optimization options: `TTICAD` (default), `EC` or `false`.

This also uses its own structured data objects, but requires a list and a declared polynomial ring as input, and is not immediately compatible with some external functions. Variable orderings can be suggested ahead of time, but the `RegularChains` package uses a decreasing variable order, interpreting $[x_1, \dots, x_n]$ as $x_1 \succ \dots \succ x_n$, rather than the more conventional $x_1 \prec \dots \prec x_n$, making cross-package comparisons more difficult. `RCCAD` avoids issues with curtains entirely and can exploit multiple ECs and even inequations when refining its decomposition.

Although the two implementations differ significantly in structure, they rely on similar algebraic tools, including squarefree factorisation, discriminants, resultants, and leading coefficients, and both reflect cutting-edge research.

5.2.1 Heuristics

The Tonks Heuristic

Tonks [Ton21] introduces a heuristic, based on Brown (see Definition 2.7.1), which takes advantage of ECs:

Definition 5.2.1 (Tonks heuristic). *For a Tarski formula Φ and $\mathcal{F} = \{f_1, \dots, f_r\} \subset \mathbb{R}[\mathbf{x}]$ the associated set of polynomials, let $\mathcal{G} \subseteq \mathcal{F}$ be the set of polynomials in $\mathcal{F}$ associated with ECs in Φ , $\hat{\mathcal{F}} = \mathcal{F} \setminus \mathcal{G}$ the set of polynomials associated with inequational constraints in Φ , and $\mathbf{x}_{\mathcal{G}}, \mathbf{x}_{\hat{\mathcal{F}}} \subseteq \mathbf{x}$ the variables in $\mathcal{G}$ and $\hat{\mathcal{F}}$ respectively. The Tonks heuristic suggests a variable ordering by applying each step of the Brown heuristic, first to $\mathcal{G}$, then to $\hat{\mathcal{F}}$, that is:*

- *lowest degree in $\mathcal{G}$, with $\deg_{v \notin \mathbf{x}_{\mathcal{G}}}(\mathcal{G}) = \infty$,*
- *lowest degree in $\hat{\mathcal{F}}$,*
- *lowest total degree among the terms in $\mathcal{G}$ in which the variable appears,*
- *lowest total degree among the terms in $\hat{\mathcal{F}}$ in which the variable appears,*
- *lowest number of terms in $\mathcal{G}$ containing the variable,*
- *lowest number of terms in $\hat{\mathcal{F}}$ containing the variable.*

This heuristic prioritises variables within ECs in order to leverage restricted projections as much and as early as possible, taking as much advantage of ECs as possible.

The dsum Heuristic

During initial testing, it was noted that for many examples the Brown heuristic gave a very poor ordering. Instead, we looked to use the ordering **dsum**, inspired by **gmods**. This heuristic simply orders the variables by their degree sum D_{x_k} (see Section 2.7), but differs from **gmods** by determining the full ordering at the start, based on the input polynomials. This is because the idea of determining the next variable after projecting does not translate to the incremental RC-CAD algorithm, with no way to calculate this ordering in a representative way. Regardless, there is currently no way to use the **gmods** heuristic with PLCAD without first determining the full projection order manually.

5.2.2 Comparison

It is important to acknowledge that a truly fair comparison between PLCAD and RCCAD is not possible for various reasons.

Firstly, the PL-CAD approach is much older, and consequently has benefited from greater attention, optimisation and documentation. In contrast, RC-CAD is more recent, and while it adopts several ideas from PL-CAD, including the use of ECs and TTICADs, it also uses features which PL-CAD cannot, warranting further exploration.

In terms of software, the opposite is true: the **RegularChains** package is older and has been developed over a longer time period, while the **QuantifierElimination** package (and the Lazard theory that underpins it) is much newer.

One large barrier we do not have to contend with is differing systems: both packages are implemented in *Maple*, making direct comparisons much more straightforward. However, there is the consideration of *how* they are implemented: one algorithm may be more efficiently implemented than the other in terms of features, code efficiency and integration with other packages. Furthermore, this comparison is also concerned with the current implementations, so any results may not necessarily imply one method is better than the other.

Despite this, such a comparison is still valuable: if RC-CAD performs competitively despite being much newer, this suggests the promise of the **RegularChains** framework and highlights opportunities for further research.

To make the comparison as fair as possible, both algorithms are run on the same dataset, and use the same features when possible. They are tested on equivalent formats of the input formula, as well as on a list of its polynomials. The command **QuantifierElimination:-QuantifierTools:-GetAllPolynomials** was used to extract the polynomials from the input formula, and was performed before timing experiments started.

For both cases, three sets of comparisons are run, two using a fixed ordering decided externally, and one chosen by that package’s default heuristic. External orderings are determined before timings started, while each package’s default heuristic is included in the timing to benchmark their implementation.

Another practical obstacle is that the two algorithms require and output data in different formats, meaning many external and internal commands are not universally compatible. PLCAD can directly interpret the Tarski formulae from the dataset, while RCCAD requires input as lists in disjunctive normal form (DNF) (see point 2 below). For fairness, the input formula is first translated to DNF, then to a suitable format for each package, ensuring the formats match as close as possible before timing measurements begin.

5.2.3 Issues and Solutions

Due to the different structures and implementations of the two packages, various issues and challenges were encountered. These issues and their resolutions are detailed below.

1. Both packages contain a command named `CylindricalAlgebraicDecompose`, and the name `QuantifierElimination` refers to both the package itself and a command in `RegularChains`. Specifically, once `RegularChains` is loaded, calls to `QuantifierElimination:-CylindricalAlgebraicDecompose` are misinterpreted. This can be resolved by instead calling each package's command directly, or by using the alias `QE:=QuantifierElimination`, allowing the function to be called unambiguously as `QE:-CylindricalAlgebraicDecompose`.
2. The input formulae from the dataset are stored as quantifier-free Tarski formulae, which `RCCAD` cannot work with, instead requiring its input as a DNF-like list structure. To ensure a fair comparison, a custom converter was developed to take the input formula in DNF and translate it to a `RegularChains`-amenable list structure. For example:

- $(A = 0) \vee (B > 0 \wedge C \leq 0)$ becomes $[[A = 0], [B > 0, C \leq 0]]$,
- $\dots(A \wedge B)\dots$ becomes $[\dots[A, B]\dots]$,
- $\dots(A \vee B)\dots$ becomes $[\dots[A], [B]\dots]$.

In general, conjunctions within a disjunct are grouped into a single list, and disjunctions correspond to separate lists.

Additionally, converting a formula to DNF using standard packages makes it incompatible with `PLCAD`, so an additional command was made to address this. This ensures both packages are given logically equivalent input, with timing experiments starting only after all conversion steps are complete.

3. `QuantifierElimination` does not contain a command which allows the user to export the decided variable ordering either before or after CAD construction. The `QuantifierElimination` command `PrintProjection` was used to print the projection polynomials at each level, from which the variable order can be inferred. Since this command only prints data rather than returns it, all *Maple* workings were output to a text file, which was then processed by an external script to reconstruct the ordering.
4. It was initially unclear whether `RCCAD` returned only cells satisfying at least one input constraint, which could have affected cell-count comparisons. Upon testing, this was found not to be the case, and so no correction was necessary.

5.3 Setup and Layout

Benchmarking was performed on a PC Specialist NH5x_7xDCx_DDx laptop running Microsoft Windows 10 Home, equipped with 32 GB of RAM and an Intel Core i7-1165G7 CPU at 2.20 GHz, using *Maple* 2024.2.

Experiments were automated through a series of Shell and *Maple* scripts in order to ensure reproducibility and consistency across tests. Each formula in the dataset is stored as a `.smt2` file, which was processed by a Shell script invoking the command-line *Maple* environment (`cmaple`), which parsed the input formula, ran the selected CAD command, and recorded time, memory and cell count. The key components of this were:

1. `run_all_n.sh`: The main script file, handling the overall testing: it prepares a timestamped logfile and iterates over every input file, sending it to `cmaple` and running the *Maple* script `process_file_n.m`, recording the output to the logfile.
2. `process_file_n.m`: The main *Maple* file. This first uses `SMTLIB:-ParseFile` to load the formula, and loads custom commands from `TimingsCommands.mpl`. It then converts the formula into an appropriate format, and determines a variable ordering if it can be supplied independently. When each package's own ordering heuristics are used, the ordering is determined as part of the main CAD command, to ensure fairness. It then attempts a test run of each CAD command using the supplied data, ordering and options, returning the associated error or timeout if unsuccessful, and continuing to the main test if successful. This test runs the same CAD command using `CodeTools:-Usage` to capture the average memory, CPU time, and real time over ten iterations.
3. `TimingsCommands.mpl`: A file containing the following: `DegreeSumHeuristic`, which orders variables depending on the degree sum of the input polynomials, and the commands `ConvertToPL` and `ConvertToRC`, which take a DNF formula obtained using the `Logic` package, and puts it into a form usable by the `QuantifierElimination` and `RegularChains` packages, respectively.
4. **Logfile**: This is simply the output of `run_all_n.sh`, allowing the user to check in more detail and look for any additional issues and record additional data not captured directly by *Maple*.
5. `Log_to_csv.sh`: A conversion tool, used to extract key data from the logfile and neatly display it ready for analysis. This also reconstructs the variable ordering chosen by PLCAD when using the Tonks heuristic.

6. **Results.xlsx**: A template used to aggregate and compare results from each `.csv` file, highlighting best and worst timings, differences and ratios, failure counts and producing survival plots.

Each test was run with a 240-second timeout, using `CodeTools:-Usage` to record time and memory averaged over ten iterations. Only the CAD construction is measured in order to ignore time taken up by preprocessing, input conversion, and *Maple* initialisation. When an ordering is predefined, it is computed before CAD construction begins, while each package’s default ordering is computed during CAD construction, serving to measure the strength and implementation of each package’s default ordering heuristic.

As *Maple*’s `CodeTools:-Usage` and `timelimit` commands cannot be used together to interrupt individual iterations within such a batch, an initial test run was used to detect errors or timeouts, only proceeding to the main ten-iteration test if successful. While this is not ideal, it avoids extended runtimes for examples known to always error or time out, and avoids compounded issues where `timelimit` is not able to abort immediately during a process.

5.4 The Dataset

We conducted a series of experiments to compare the performance and behaviour of the two implementations, looking at the effects of specific options and choices, including the use of ECs, variable ordering heuristics, and pre-processing options. Table 5.1 summarises the configuration of each test.

Experiments were run on a modified version of the `metitarski-polypaver-3vars` dataset from SMT-LIB¹, which consists of problems in three variables stemming from industry [JdM12], and originating as proof obligations in the *MetiTarski* project [JdM12, AP10]. While criticisms have been made about whether this is truly a representative dataset in terms of heuristic tuning [HHP⁺23, JD24], they do not affect the validity of the dataset for our purposes.

As is the case with most of *MetiTarski*, the original dataset contained mostly inequalities ($\leq$), with no exploitable ECs. Single ECs were artificially added to most examples by replacing the block $-1/32 \leq E \wedge E \leq 1/32$ with $E = 0$. This allowed each algorithm to make use of a single EC, but meant each package’s default options (PLCAD exploiting multiple ECs and RCCAD using TTICAD) could not be tested in full generality and therefore these options were not used.

¹`QF_NRA.tar\non-incremental\QF_NRA\meti-tarski\polypaver\sqr43-int\3vars`, found in <https://zenodo.org/records/15493096>, from <https://smt-lib.org/>.

Tests were run first on a simplified list of polynomials obtained from the formula (in order to test each algorithm without tools or shortcuts), then on the formula in DNF (in order to leverage the available EC). For both of these cases, the algorithms were tested on different orderings: a fixed lexicographic ordering, each package’s default ordering (both variants of Brown), and another ordering heuristic inspired by `gmods`.

5.5 The Tests and Results

The dataset consists of 67 `.smt2` files, each containing a quantifier-free Tarski formula written in SMT-LIB language, which can be read by *Maple* using the SMTLIB package. Although the dataset is not entirely balanced in terms of variable order, it contains real-world examples which use Boolean connectives and constraints, allowing each implementation to make use of ECs, something other datasets lack.

Each problem was run under six configurations, combining different treatments of the input, variable order, and equational constraint handling. The three cases for variable ordering heuristics were:

- **Fixed variable ordering:** Variables are ordered lexicographically.
- **Default heuristic ordering:** Each package selects its own (default) variable ordering. `QuantifierElimination` uses the EC-aware Tonks heuristic, while `RegularChains` uses `SuggestVariableOrder(F, decomposition = cad)`. Both are derived from the Brown heuristic, so broadly similar behaviour is expected in the polynomial-only case.
- **Degree sum ordering:** Inspired by `gmods`, variables are ordered by the sum of their degree in the given polynomials. Unlike `gmods`, this is only determined using the input polynomials, but has been shown to give a superior ordering to Brown in various examples.

These three orderings were run on two forms of the input data:

- **Polynomials only:** The initial problem is stripped using the command `QuantifierElimination:-QuantifierTools:-GetAllPolynomials(F)` to a list of polynomials without constraints, thereby testing the underlying algorithms without any shortcut gained from EC or structure exploitation.
- **Full formula:** The Tarski formula is first converted to DNF and then the appropriate format, ensuring both implementations handle a logically identical input and can make use of ECs. Here `PLCAD` uses the option `useequations=single`, while `RCCAD` uses `optimization=EC`.

Each configuration tests a distinct combination of algorithmic and heuristic features, including a ‘worst case’ (no ECs, fixed order) and each package’s default ordering.

	PLCAD (QuantifierElimination)	RCCAD (RegularChains)
Polynomials	Input: <code>convert(QE:-QuantifierTools:-GetAllPolynomials(import), list)</code>	
	Format: <code>input</code> Ordering: <code>variablestrategy=qe_ordering</code> Options: <code>propagateecs=false,</code> <code>useequations=none, usegroebner=false</code> Cell count: <code>QE:-CADDData:-</code> <code>NumberOfLeafCells(output)</code>	Format: <code>input</code> Ordering: <code>RC:-PolynomialRing(rc_ordering)</code> Options: <code>output=allcell,</code> <code>optimization=false</code> Cell count: <code>nops(output)</code>
Full formula	Input: <code>Normalize(QE:-QuantifierTools:-ConvertToPrenexForm(import), form=DNF)</code>	
	Format: <code>ConvertToPL(input)</code> Ordering: <code>variablestrategy=qe_ordering</code> Options: <code>propagateecs=false,</code> <code>useequations=single, usegroebner=false</code> Cell count: <code>QE:-CADDData:-</code> <code>NumberOfLeafCells(output)</code>	Format: <code>ConvertToRC(input)</code> Ordering: <code>RC:-PolynomialRing(rc_ordering)</code> Options: <code>output=allcell,</code> <code>optimization=EC</code> Cell count: <code>nops(output)</code>

Table 5.1: Configuration options for each CAD command.

5.6 Results and Observations

The results of these experiments are plotted as *cactus plots* [BDG17], measuring aggregate (real) time taken against the number of successful tests. The time taken for each successful test are collected and ordered from smallest (t_1) to largest (t_k), and $\sum_{i=1}^k t_i$ (using a logarithmic scale) is plotted against i , where k is the total number of tests. This allows one to easily compare the time taken and number of successes between each method and set of options.

We compare the performance of PLCAD and RCCAD using each variable ordering, first on the polynomials obtained from the formulae, then on the DNF form, wherein ECs can be exploited. We then explore the impact of this EC exploitation by comparing the results for each PLCAD and RCCAD configuration. For each experiment we also provide a breakdown of successes, timeouts and errors, the number of times one implementation was faster (both including and excluding cases where the rival implementation errored or timed out), and the number of times one algorithm produced fewer cells, with the same breakdown.

We then compare every configuration in a final compendium plot (Figure 5-6), where these results are additionally plotted against a *virtual best* (VB), whose time taken for each test is the minimum among the results. This is used to track how close each configuration is to the best performance, especially at different thresholds, and helps to identify the strengths and weaknesses of each.

5.6.1 Lex Ordering, Polynomials Only

RCCAD performed substantially better. PLCAD often timed out or errored, except on simpler examples. For the simplest examples, PLCAD was faster and used less memory overall (on average: 66.738% of the memory, 91.075% of the CPU time, and 90.334% of the real time compared to RC). However, these small gains may be attributed to PLCAD's lower setup overhead and simpler projection phase.

For these simple examples, cell counts were identical. In the small number of more difficult cases, RCCAD produced significantly fewer cells than PLCAD (9 vs. 57).

Of the cases where both succeeded:

- PLCAD used 69.089% of the memory used by RC, and achieved modestly faster runtimes.
- RCCAD produced slightly fewer cells overall.

Success Rate:

RCCAD: 45 Successes (67.16%), 21 Timeouts (31.34%), 1 Errors (1.49%)

PLCAD: 14 Successes (20.90%), 21 Timeouts (31.34%), 32 Errors (47.76%)

No. times faster (realtime):

RCCAD: 35 (52.24%), 4 actual (5.97%)

PLCAD: 10 (14.93%), 10 actual (14.93%)

No. times fewer cells:

RCCAD: 35 (52.24%), 4 actual (5.97%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

5.6.2 Heuristic Ordering, Polynomials Only

Both algorithms performed better under heuristic ordering. PLCAD succeeded more often than with lex ordering, providing more comparisons. RCCAD consistently produced equal or fewer cells. As expected, both chose the same variable ordering, as no ECs were available. Of the cases where both succeeded:

- PLCAD again performed better on simpler examples, while RCCAD performed better on more difficult ones.
- In both cases, the difference was substantial, with the winner taking around half the time or memory of the loser in many cases.
- Overall, RCCAD used slightly less memory, but significantly less CPU and real time compared to PLCAD.
- RCCAD achieved a very small reduction in cell count.

These results suggest that RCCAD may require slightly more setup time and memory but provides significant performance benefits.

Success Rate:

RCCAD: 62 Successes (92.54%), 5 Timeouts (7.46%), 0 Errors (0.00%)

PLCAD: 20 Successes (29.85%), 3 Timeouts (4.48%), 44 Errors (65.67%)

No. times faster (realtime):

RCCAD: 52 (77.61%), 10 actual (14.93%)

PLCAD: 10 (14.93%), 10 actual (14.93%)

No. times fewer cells:

RCCAD: 52 (77.61%), 10 actual (14.93%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

5.6.3 Degree Sum Ordering, Polynomials Only

Nearly every test succeeded. PLCAD experienced two timeouts on tricky examples, but was otherwise only proportionally slower than RCCAD.

When both succeeded, as expected, PLCAD was faster for the simplest examples, while RCCAD was faster for the rest. PLCAD had a strong advantage in memory usage (85.550% that of RC), whereas RCCAD had a comparable advantage in total runtime (85.561% CPU time, 88.917% real time). Cell counts were relatively even, with RCCAD always having the same or slightly fewer cells.

Success Rate:

RCCAD: 67 Successes (100.00%), 0 Timeouts (0.00%), 0 Errors (0.00%)

PLCAD: 65 Successes (97.01%), 2 Timeouts (2.99%), 0 Errors (0.00%)

No. times faster (realtime):

RCCAD: 57 (85.07%), 55 actual (82.09%)

PLCAD: 10 (14.93%), 10 actual (14.93%)

No. times fewer cells:

RCCAD: 57 (85.07%), 55 actual (82.09%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

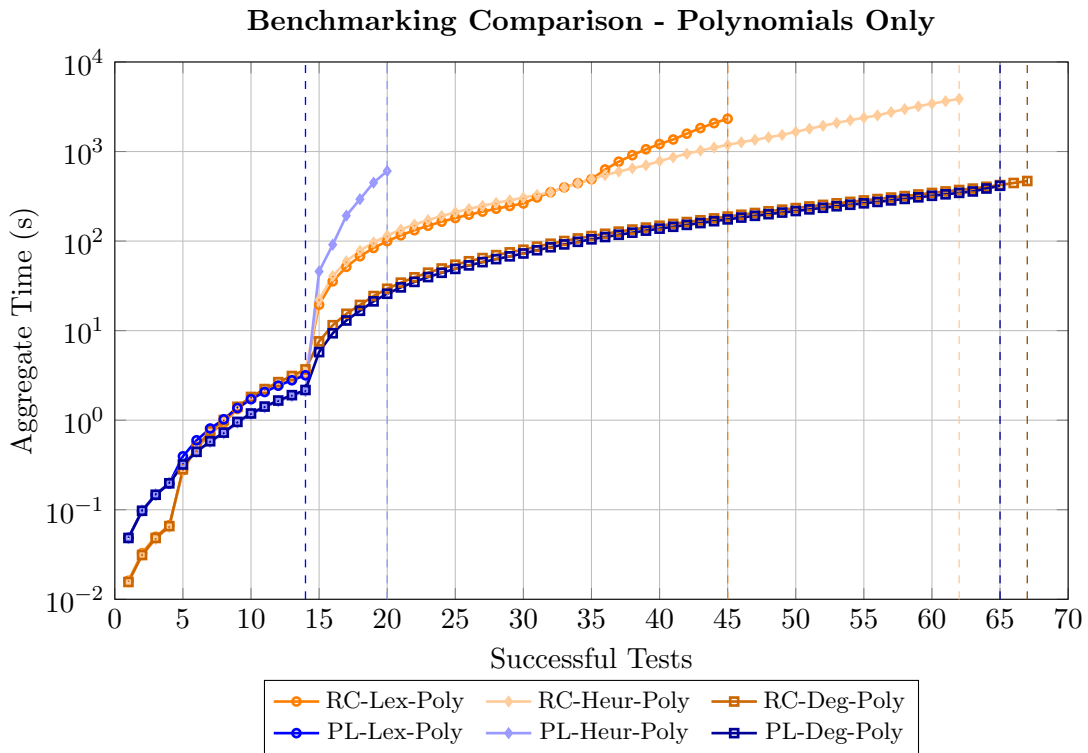


Figure 5-2: Cactus plot comparing the performance of PLCAD and RCCAD on collections of polynomials, using different variable orderings: lex, heuristic (Brown) and `dsum`.

5.6.4 Lex Ordering, Exploiting the EC

For PLCAD, results were similar to those conducted on just polynomials, only succeeding on simple examples with very little difference in time or memory. RCCAD had a much more pronounced difference, taking around half the time and memory than before, and almost a third of the cells, due to being able to leverage the EC and any inequations

present to avoid irrelevant computation.

Both methods succeeded in a similar number of cases, with **PLCAD** failing on the same examples as before, and **RCCAD** succeeding on some earlier examples it failed on before, but timing out on some later ones it didn't before. When both succeeded, **RCCAD** outperformed across the board:

- 75.070% of the memory,
- 53.221% of the CPU time, 53.970% of the real time, and
- 32.654% as many cells.

Success Rate:

RCCAD: 46 Successes (68.66%), 20 Timeouts (29.85%), 1 Errors (1.49%)

PLCAD: 14 Successes (20.90%), 21 Timeouts (31.34%), 32 Errors (47.76%)

No. times faster (realtime):

RCCAD: 47 (70.15%), 14 actual (20.90%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

No. times fewer cells:

RCCAD: 47 (70.15%), 14 actual (20.90%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

5.6.5 Heuristic Ordering, Exploiting the EC

Significantly fewer failures for **PLCAD**, with **RCCAD** failing on the same tests due to timeout. **PLCAD** has a much more significant improvement than **RCCAD** compared to its polynomial-only version, likely due to selecting a better variable ordering. Despite this, it still performs significantly worse than **RCCAD**, even when each heuristic chooses a different ordering. Among the 18 out of 67 cases where the variable ordering matched,

- **PLCAD** was better in terms of memory (88.961%),
- **RCCAD** was significantly faster in terms of CPU time (56.415%) and real time (76.175%), and
- **RCCAD** had a modest advantage in cell count.

In cases where the variable ordering differed, the difference was dramatic, with **RCCAD** taking only 37.615% of the memory, 26.164% of the CPU time, 28.031% of the real time, and 15.632% of the cells that **PLCAD** did.

Success Rate:

RCCAD: 62 Successes (92.54%), 5 Timeouts (7.46%), 0 Errors (0.00%)

PLCAD: 47 Successes (70.15%), 0 Timeouts (0.00%), 20 Errors (29.85%)

No. times faster (realtime):

RCCAD: 52 (77.61%), 37 actual (55.22%)

PLCAD: 10 (14.93%), 10 actual (14.93%)

No. times fewer cells:

RCCAD: 56 (83.58%), 41 actual (61.19%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

5.6.6 Degree Sum Ordering, Exploiting the EC

This test showed a similar but less extreme trend than the heuristic ordering, with both performing very competitively. Every test succeeded, with:

- PLCAD using less memory (80.211% of what RCCAD used),
- RCCAD taking less time (91.974% of the CPU time and 94.862% of the real time of PLCAD),
- RCCAD producing 88.697% of the cells of PLCAD.

In terms of raw wins, the two were very competitive, having a close to even split for memory, CPU time and realtime.

Success Rate:

RCCAD: 67 Successes (100.00%), 0 Timeouts (0.00%), 0 Errors (0.00%)

PLCAD: 67 Successes (100.00%), 0 Timeouts (0.00%), 0 Errors (0.00%)

No. times faster (realtime):

RCCAD: 35 (52.24%), 35 actual (52.24%)

PLCAD: 32 (47.76%), 32 actual (47.76%)

No. times fewer cells:

RCCAD: 61 (91.04%), 61 actual (91.04%)

PLCAD: 0 (0.00%), 0 actual (0.00%)

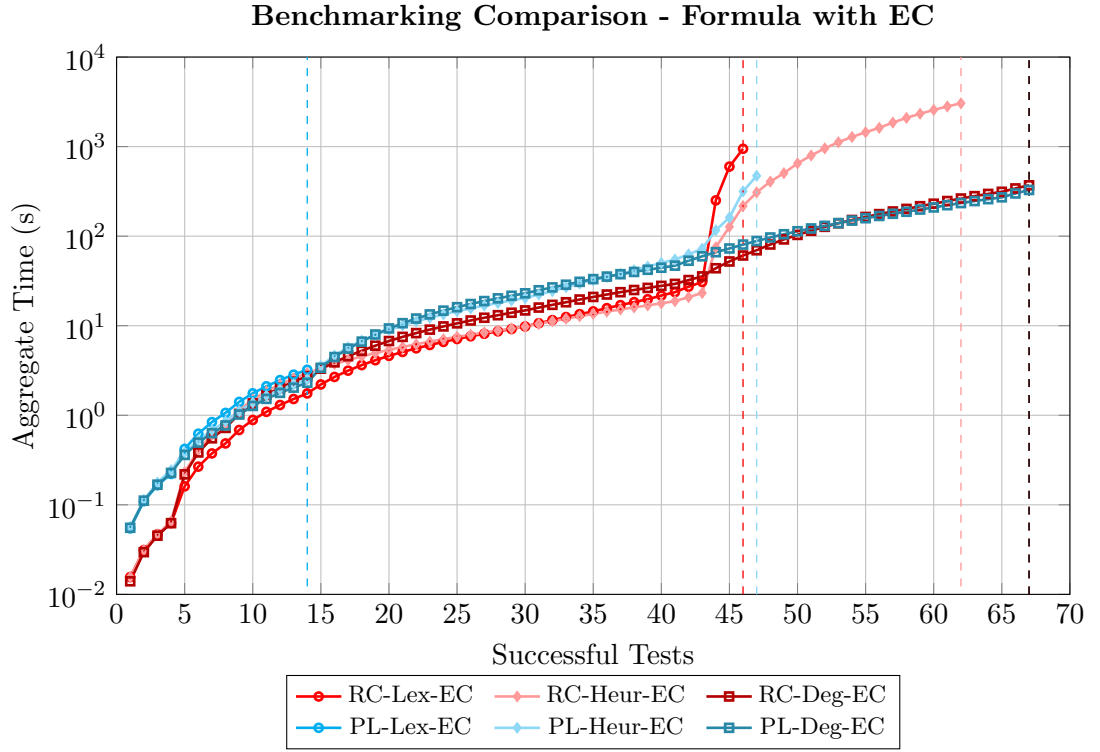


Figure 5-3: Cactus plot comparing the performance of PLCAD and RCCAD on DNF formulae containing ECs, using different variable orderings: lex, heuristic (Brown for RCCAD, Tonks for PLCAD) and dsum.

Tonks/Brown vs. dsum for tests on full formulae:

	No. times better	Total improvement (s)
PLCAD (Tonks)	16	3.5616
PLCAD (dsum)	51	334.4977
	No. times better	Total improvement (s)
RCCAD (Brown)	31	11.4695
RCCAD (dsum)	36	2009.8977

Table 5.2: A table comparing the number of times one ordering outperformed the other for each algorithm, with the cumulative time difference across those cases.

The `dsum` ordering generally performs much better than the Tonks/Brown ordering on this dataset. Both orderings take significantly less time on more difficult tests than with lex ordering, but notably RCCAD with Brown was slightly ($< 0.5s$) faster than RCCAD

with `dsum` for many (31) earlier examples. On more difficult examples, `RCCAD` with `dsum` outperforms Brown by a large margin, but `PLCAD` with `dsum` often outperforms `RCCAD` with `dsum` by around a second on average.

The `dsum` ordering was better overall, producing better timings more frequently and by a larger margin. When the Tonks/Brown heuristic did perform better, the advantage occurred only in smaller examples, giving little net benefit. This illustrates an important point worth considering: how often one implementation performs better than the other is much less important of a consideration than by how much.

5.6.7 General Observations

These results suggest that `RCCAD` performs much better than `PLCAD` when using a poor variable ordering, whereas with a good ordering the two methods are much more competitive.

The overall results seem to suggest that:

- `PLCAD` is faster only on very simple problems, likely due to its lower setup cost compared to `RCCAD`.
- Improved variable orderings and EC exploitation both led to major performance gains, and employing both simultaneously yields the greatest benefit.
- Exploiting the EC tends to significantly reduce runtime for `RCCAD`, but still causes longer experiments to time out. Improved variable orderings led to a greater number of successes, but a less dramatic reduction in runtime compared to EC exploitation.
- For `PLCAD`, exploiting the EC has little effect when used with a poor variable ordering, but becomes highly effective when the ordering is improved.

When working only with polynomials, `RCCAD` and `PLCAD` with the `dsum` ordering were very closely competitive. When instead considering the whole formula, `PLCAD` with the `dsum` ordering performed slightly better than `RCCAD` with `dsum`. `PLCAD` with the Tonks ordering also performed competitively, due to providing a better ordering than Brown.

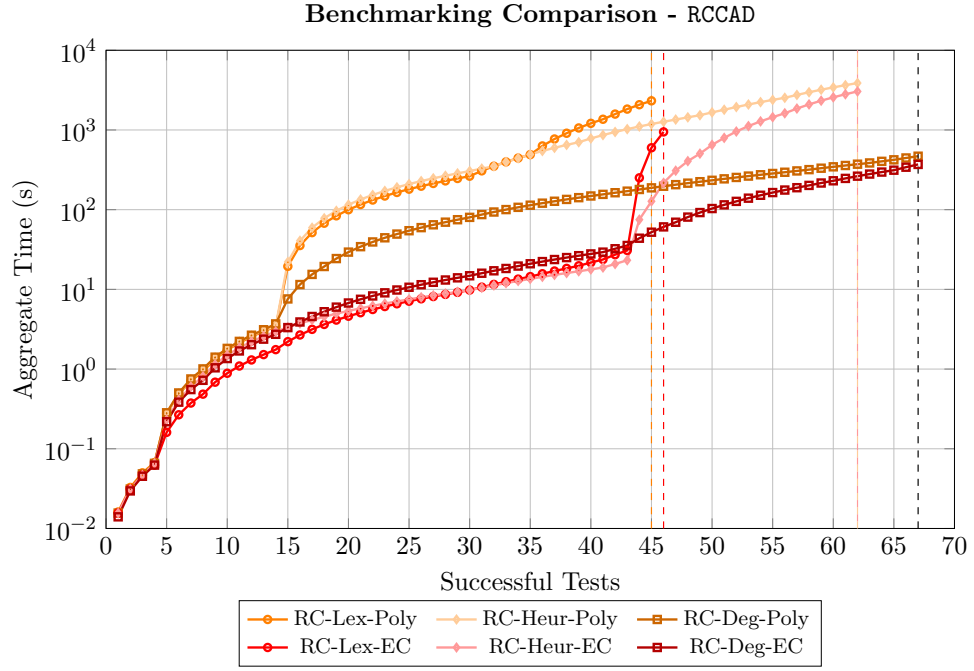


Figure 5-4: Cactus plot comparing the performance of RCCAD on both polynomials and on DNF formulae containing ECs, using different variable orderings: lex, heuristic (Brown) and `dsum`.

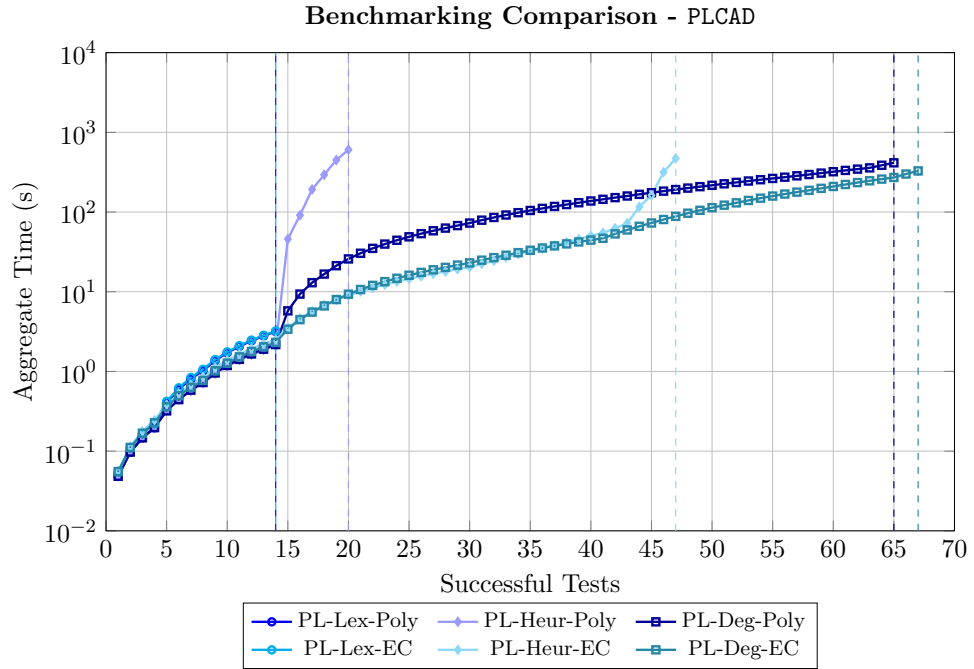


Figure 5-5: Cactus plot comparing the performance of PLCAD on both polynomials and on DNF formulae containing ECs, using different variable orderings: lex, heuristic (Brown/Tonks) and `dsum`.

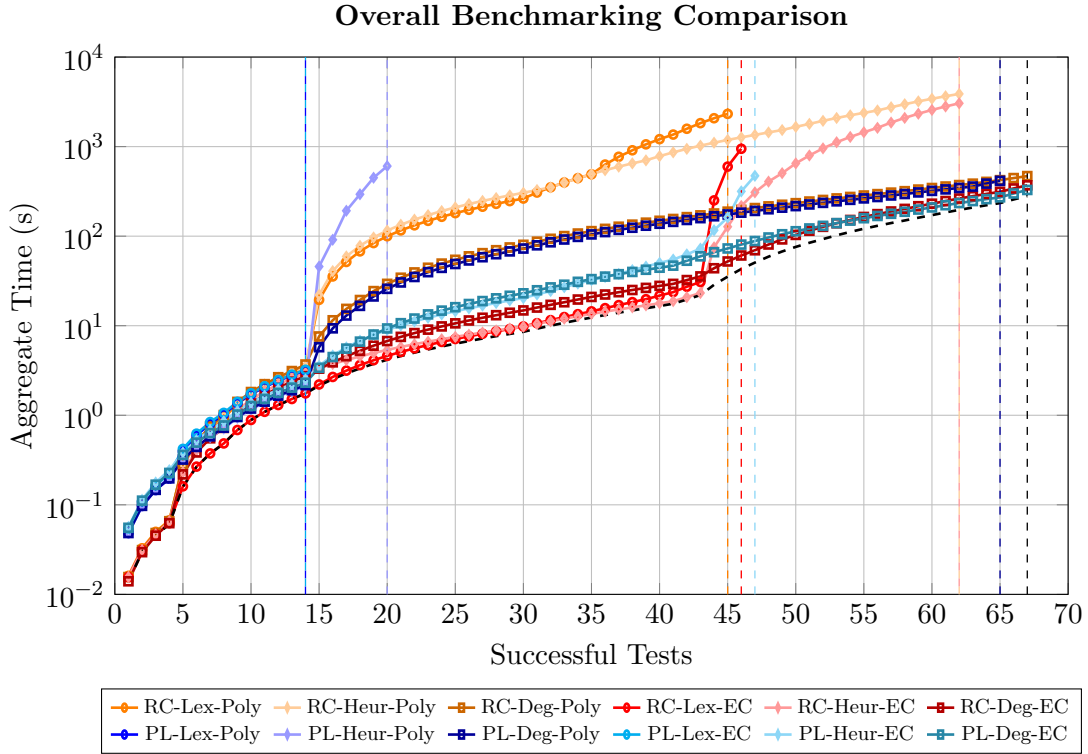


Figure 5-6: Cactus plot comparing each combination for PLCAD and RCCAD. Virtual best (VB) is plotted as a black line.

When considering cells, RCCAD with Brown and EC had the fewest in 29 cases, followed by RCCAD with `dsum` (both formats) in 24 cases, RCCAD with lex and EC in 10 cases, with four simple cases tying for cell count across each configuration.

When considering memory, PLCAD on polynomials with the Tonks ordering used the least memory for smaller examples, while RCCAD using an EC with the Brown ordering used the least for most intermediate examples, and PLCAD using an EC with Tonks or `dsum` using the least memory for the hardest examples.

It is also worth noting that PLCAD can operate directly on the input formula, without DNF conversion, and performs significantly better without it. Without conversion, PLCAD succeeded in every example using both Tonks and `dsum`, and outperformed RCCAD by a wide margin in terms of memory, time and cell count. This suggests that the DNF requirement for RCCAD may significantly limit its performance, and PLCAD is largely preferable when examples are in a more efficient format. A worthwhile direction for future work would be to extend PLCAD to work with such formats.

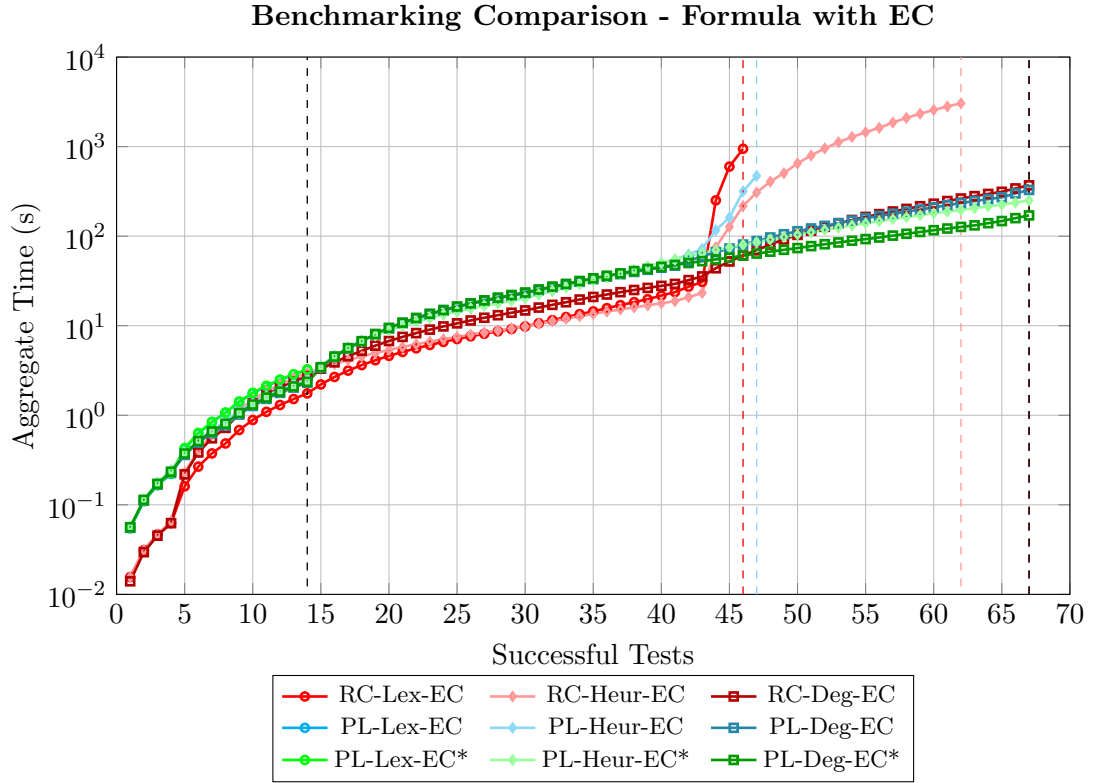


Figure 5-7: Cactus plot comparing the performance of PLCAD and RCCAD on DNF formulae containing ECs against PLCAD on formulae not converted to DNF (in green) using the three orderings.

In general, choosing between the two methods largely depends on the input and the information available. If an efficient variable ordering can be guaranteed, both methods will perform comparably, but for cases when a sub-optimal ordering may arise (due to lack of information or insight on the problem), RCCAD tends to perform noticeably better. If the input is a list of polynomials, RCCAD tends to have the edge due to its lower cell count and avoidance of redundant computation. However, for Tarski formulae, PLCAD can perform noticeably better, especially if the formula is not already in DNF. If a user wishes to prioritise fewest cells, RCCAD will always be best; if instead a user prioritises minimal memory usage, PLCAD may be preferable.

5.7 Errors

Here we summarise the *Maple* errors encountered during the benchmarking experiments. These issues are currently under investigation. It is worth noting that **RegularChains** is a bespoke package built from the ground up, whereas **QuantifierElimination** relies on several pre-existing root isolation commands, which may not have been designed with CAD in mind and may therefore struggle with the demands placed on them.

Three main errors arose for PLCAD:

Error, (in evala/Factors) reducible RootOf detected. Substitutions are %1

This error usually occurs within a second of calculation start, and suggests multiple possible substitutions is the presence of a reducible root. For this use case, a bespoke command may be required to return these substitutions, or avoid the issue altogether.

Error, (in RootFinding:-RSGateway:-try_refine_iso_tri) invalid input: numelems expects its 1st argument, t, to be of type indexable, but received FAIL[1]

This issue occurs much deeper into computation, often after several minutes. This may be due to breakdowns caused by too many layers or recursion or nested roots.

Error, (in RootFinding:-RefineRoot) attempt to refine root to target precision %1 failed with last working precision %2

This error occurs after a few minutes of computation. This inability to refine a root to a given precision is again likely due to recursion or nested roots.

RCCAD only had one error, ‘System error, ’, “bad id”. This occurred after a significant amount of time and did not happen on previous runs, appearing to be an internal *Maple* error, likely caused by specific computational or memory conditions.

Chapter 6

GraphCAD - A *Maple* Tool for Visualising 2D CADs

The `GraphCAD` package was initially developed for personal use, but has been shown to be of wider interest, having already been used by the community [Eng24]. This tool has been formally published and is available at [Lee25a], and figures obtained using this tool have been used throughout this thesis, particularly Figures 2-1, 2-3 and 4-5.

6.1 Introduction

In this chapter we present `GraphCAD`, a package which allows the user to display two-dimensional CADs in *Maple*, using either `RegularChains` or `QuantifierElimination` to produce a CAD as input. The main function, the namesake `GraphCAD`, takes a system of polynomials $\mathcal{F}_2 \subset \mathbb{R}[x_1, x_2]$ and produces a colour-coded graph representing an $\mathcal{F}_2$ -(sign-)invariant CAD of $\mathbb{R}^2$. When presented with polynomials of higher dimension, it first produces the 2D projection polynomials $\mathcal{F}_2$ and then produces the CAD graph, using an appropriate method depending on the choice made.

The package contains various other sub-commands which may be of use to the user, including a command for converting an input list, set or Tarski formula into a flat list of polynomials without constraints; a command for producing an irreducible basis; and implementations of the Brown heuristic and Lazard projection. Additionally, it allows the user to partially specify a variable ordering and obtain projection polynomials via the `RegularChains` package. Users are also able to supply their own CADs as input for the command `GraphFromCAD`.

The titular command **GraphCAD** combines the features of previous commands, taking a set of polynomials as input and first producing a set of 2D projection polynomials $\mathcal{F}_2$ using an optional partially specified order alongside an $\mathcal{F}_2$ -invariant CAD of $\mathbb{R}^2$. This command allows for the projection and CAD computation via either projection and lifting (via **QuantifierElimination**) or the incremental algorithm from **RegularChains**, and also allows one to produce an open CAD, sample points, and display the 1D CAD below.

Users are able to choose the projection and display order, and cells are graphed by dimension: 2D cells are coloured with various colours, 1D cells are represented by black lines, and 0D cells are represented by red diamonds.

The tool allows users to:

- Visualise the structure of a 2D CAD directly in Maple.
- Compare variable orderings.
- Inspect the geometric meaning of projection and lifting in small examples.

The package acts as a ‘docking station’ between two different CAD implementation and works as a visual representation suitable for inspection or publication.

There are two packages this work builds upon: the package **QuantifierElimination** [Ton21], which implements projection and lifting-style CAD using the Lazard projection, and **RegularChains**, which uses the incremental algorithm described in Section 2.9.4. As the goal of this package is to only produce and graph $\mathcal{F}_2$ -(sign-) invariant CADs of $\mathbb{R}^2$, many simplifications and improvements such as the use of ECs and TTICADs have been disabled. Within this package, and throughout this chapter, we will refer to the use of the Lazard projection, **QuantifierElimination** and projection and lifting methods in general as PL, and the incremental algorithm and **RegularChains** package as RC. That is, producing a graph of the 2D projection of $\mathcal{F}$ using `method=PL` will project the polynomials using the Lazard projection and return the output of **QuantifierElimination:-CylindricalAlgebraicDecompose**. Producing a graph of the 2D projection of $\mathcal{F}$ using `method=RC` first constructs a CCD using the **RegularChains** command **ConstructibleSetTools:-CylindricalDecompose** to obtain the 2D projection polynomials, before constructing a CAD using the command **RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose**.

As **GraphCAD** is built on top of these existing packages and does not interface with the underlying code, there is naturally room for improvement and optimisation.

6.2 GraphCAD Procedures

The package two main groups of commands: preprocessing and graphing. The former cleans and prepares input and handles projection, while the latter constructs and draws the resulting CAD.

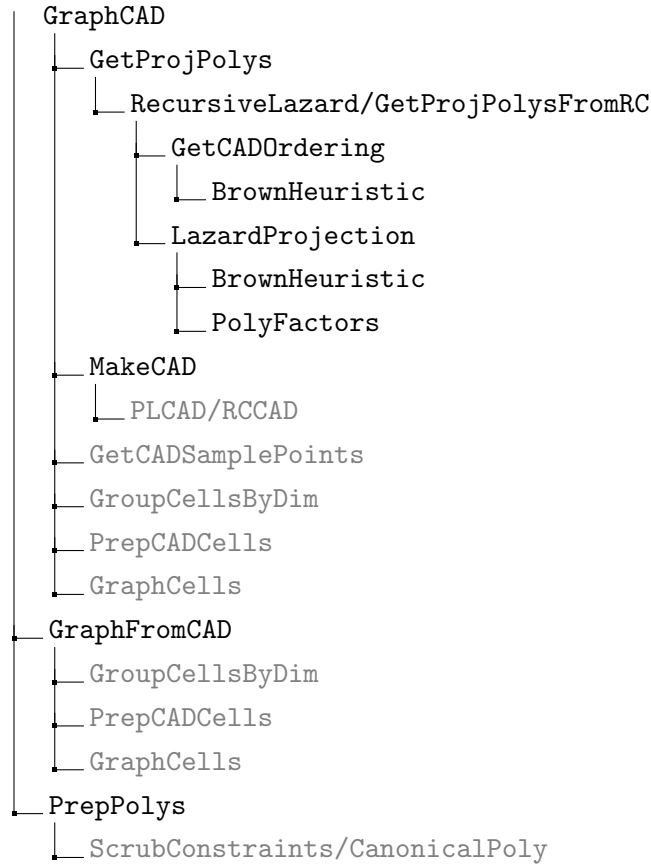


Figure 6-1: List of **GraphCAD** commands. For simplicity, internal commands (grey) are left unexpanded, while commands from other packages are omitted.

6.2.1 Pre-processing Commands

- **PolyFactors(F)**: Takes a polynomial, list, or set of polynomials and returns an irreducible basis (i.e. a list of non-constant factors).
- **LazardProjection(F, v)**: An implementation of the Lazard projection. Given a list of polynomials $\mathcal{F}$ and a variable v , it projects the irreducible basis of the polynomials with respect to v and returns the irreducible basis of the projection polynomials.

- **PrepPolys(input)**: Converts input (polynomial, constraint, list/set of polynomials/constraints, or Tarski formula) into a list of unconstrained polynomials. If the input is a Tarski formula, it makes use of the **QuantifierElimination** command **GetAllPolynomials**, and otherwise recursively flattens any nested lists or sets and converts any constraint $f \diamond \alpha, \diamond \in \{=, <, \leq, \neq\}$ into $f - \alpha$, removing constants, duplicates and unit multiples.
- **GetCADOrdering(F, vars)**: Determines a variable order among the variables in $\mathcal{F}$. If a list of variables is supplied, these variables are treated separately and may be placed at the start or the end of the list, in the order supplied.
 - **BrownHeuristic(F)**: Given a list of polynomials $\mathcal{F}$, uses the Brown heuristic (see Definition 2.7.1) to determine a projection order (with the option to additionally print the values for each variable). Polynomials are exported from lowest to highest.
- **GetProjPolys(F, vars)**: Allows the user to project $\mathcal{F}$ using either a projection and lifting method (**method** = ‘PL’) or via regular chains (**method** = ‘RC’), returning either the full list of projection polynomials or those of a specific level.
 - **RecursiveLazard(F, vars)**: Recursively applies the Lazard projection to $\mathcal{F}$ using the supplied variable order. By default, it will return the full set of projection polynomials from level 1 to level n , but can be made to stop when the supplied variables are projected away and return only a specific set of projection polynomials.
 - **GetProjPolysFromRC(F, vars)**: Obtains a set of projection polynomials from the CCT of $\mathcal{F}$ (see Definition 2.9.8 and Figure 2-6).

This preprocessing ensures that most sensible inputs can be reduced to a standard polynomial set suitable for constructing a 2D CAD. For polynomials in more than two variables, one may use **GetProjPolys** to produce a projection set in two dimensions.

6.2.2 CAD Construction and Graphing Commands

- **MakeCAD(F, vars, method)**: Produces a CAD of $\mathcal{F}$ with a variable order informed by the variables supplied, and using either **PLCAD** or **RCCAD** depending on the chosen method (‘PL’ or ‘RC’).
 - **PLCAD(F, vars)**: Constructs a CAD using the **QuantifierElimination** package, returning a **QuantifierElimination:-CADData** object which can

be passed to later commands.

- **RCCAD(F, vars)**: Constructs a CAD using the incremental algorithm from the **RegularChains** package, returning by default a list of lists of cell descriptions.

The outputs of each package are stored in different formats and require different commands. There are various internal commands which handle quirks and differences between packages and transform each output into a format suitable for graphing.

PL output requires additional work to get its output as a list of cell descriptions, and this output uses conjunctions and stores nothing for the case ‘any x ’. On the other hand, RC output stores its output as a list of sublists, with each sublist a list of constraints, including ‘ $x = x$ ’ for ‘any x ’.

The command **PrepCADCells(CAD, method)** will take the output of **MakeCAD** and produce a list of lists, in which each sublist is the description of a cell. If a method is not specified, the command will attempt to determine one based on the input format. Similarly, the command **GetCADSamplePoints(F, vars, method)** obtains and formats the CAD sample points for the given polynomials.

This cell list is passed to the command **GroupCellsByDim(cell_list)**, which separates the cells by dimension: 0D (points), 1D (curve segments), and 2D (regions), ready for the graphing commands to assign colours and line styles for the plot.

The command **GraphCells(cellslist, vars)** is the ‘core plotting engine’, taking the dimensionally-separated output and uses this to construct the colour-coded graph: 2D cells are coloured in a cyclical fashion from a given list of colours, while 1D boundaries are drawn as black lines and 0D points are drawn as red diamonds, with the additional option to show a 1D projection below the main plot.

The main two commands build upon this:

- **GraphFromCAD(CAD, vars)**: This combined the previous internal commands, taking a 2D CAD as input and producing its associated graph. The user can also visualise the induced 1D CAD below using the option **onedplot=true**.
- **GraphCAD(F, varorder, displayorder, CADmethod)**: Top-level command taking a set of polynomials $\mathcal{F}$ as input, projects this to a set of 2D polynomials $\mathcal{F}_2$ and produces the graph of the associated CAD. The user is able to specify both the projection and display order, the method used (PL or RC), the plotting ranges and

options for producing sample points (`samplepoints=true`), visualise the induced 1D CAD below (`onedplot=true`) and only displaying open cells (`opencad=true`).

In the 2D visualisations produced, 2-cells are shaded in various colours, 1-cells are drawn in black, and 0-cells are represented as red diamonds. The colours of the 2-cells are the following, taken in order, drawn left-to-right and bottom-to-top:

```
["Salmon", "CornflowerBlue", "LightGoldenrod", "LightGreen", "Plum",
"SandyBrown", "SkyBlue", "MediumPurple", "MediumAquamarine", "MistyRose"]
```

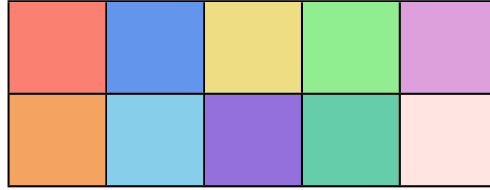


Table 6.1: The colours used to mark 2D cells.

When using the option `onedplot=true`, the 1D CAD is displayed as a line ‘below’ the plot, consisting of a line partitioned by hollow red diamonds.

Further information on the commands shown in this section, including usage, options and worked examples, is shown in Section 6.4.

6.3 Choices and Limitations

- The `LazardProjection` and `BrownHeuristic` both needed to be implemented manually, despite the former being used in `QuantifierElimination` and the latter being used by both methods. This is due to `QuantifierElimination` having a way to print projection polynomials and variable orderings but not export them, and `RegularChains` ordering variables in the opposite order (so in the case of a complete tie, variables are chosen in reverse alphabetical order).
- `GetCADOrdering` allows the user to partially specify the order of variables at input. Users are able to supply a list of polynomials, and these polynomials are removed from the ordering and placed at the start or end of the list, in the order specified. This allows for something analogous to projecting a quantified formula, as the user can specify variables to project first, and the order in which to project them. This marks a useful first step towards implementation of block orderings and block cylindricity.

- This package also marks better interactivity between the different implementations of CAD algorithms, allowing users to easily project and produce CADs using either method, allowing for easier comparisons between the two and mitigating complications caused by different inputs and syntax. For example, **RegularChains** requires a ring to be specified and lists variables in decreasing order, but both of these are handled internally, allowing the user to specify variables in the standard (increasing) order.

Some limitations of the package include:

- As expected with CAD, input of high dimension, degree, or number of polynomials will take considerable time to compute. This issue is lessened slightly by **GraphCAD**'s focus on two-dimensional CADs, avoiding the work associated with lifting to higher dimensions, but still has a significant impact. The degree and number of polynomials in the two-dimensional projection set will grow very rapidly with the complexity of the input polynomials, and will still require a huge number of cells to be computed and drawn.
- When projecting using `method=PL`, the package uses the standard Lazard projection, without taking advantage of ECs, and therefore does not benefit from the simplifications gained from the restricted and semi-restricted projections.
- Similarly, the command **GraphFromCAD** requires a CAD of $\mathbb{R}^2$ as input, and is not able to use a CAD of higher-dimensional space.
- As **GraphCAD** uses specific settings for **PL** and **RC** and therefore does not fully integrate each package, CADs produced using other settings may not function correctly as input, and other packages and methods are not supported.
- Similarly, this package also disables certain improvements such as the use of ECs and TTICADs, so will not make use of their savings and will take longer to produce.
- The package is currently set up to produce a graph of a CAD in two dimensions only, and will return an error if handed an input in only one dimension.
- In the current version of the tool, the `opencad` and `samplepoints` options cannot be used together to display only the sample points of open cells. A potential future fix would involve either matching sample points to their cells, or generating them directly from the cell descriptions, returning only full-dimensional sample points.

6.4 Example Usage in *Maple*

We conclude this chapter by demonstrating the various commands available in **GraphCAD**. As these commands have already been described in Section 6.2, commentary largely serves to demonstrate each command's usage and arguments.

This section largely follows the example worksheet provided in [Lee25a] and is presented in the style of a *Maple* worksheet, with inputs in italics, outputs in blue, and information, errors, and warnings in monospaced font. Inputs are delimited by semicolons, with output suppressed if a colon is used instead. The symbol ‘%’ is used to call the output of the previous command, and the operator ‘:=’ denotes assignment.

with(GraphCAD);

*[BrownHeuristic, GetCADOOrdering, GetProjPolys, GetProjPolysFromRC,
GraphCAD, GraphFromCAD, LazardProjection, MakeCAD, PolyFactors,
PrepPolys, RecursiveLazard]*

PolyFactors(expand(12(x - 1)²(x - 5)(x - y)⁶(y - z)(1 - z)));

[z - 1, y - z, x - 1, x - 5, x - y]

PrepPolys is able to handle Tarski formulae and nested lists and sets, with or without constraints:

PrepPolys(X(1 - $\frac{X}{4}$) ≤ R ∧ E ≤ $\frac{1}{32}$ ∧ - $\frac{1}{32}$ ≤ E ∧ X ≤ 2 ∧ R ≤ 3 ∧ $\frac{1}{2}$ ≤ X ∧ 0 ≤ R);

PrepPolys([[x = 1, y < 2x], [x = 1, y = 2x], [x = 1, y > 2x]]);

PrepPolys([(x - 1) · (y + 2) · (-1), (x - 1) · (y + 2) · 2]);

PrepPolys([B, [C, [c, d], e], f, [G, c, h]]);

PrepPolys([e, f, {c, d}]);

[-R, -1 - 32E, 1 - 2X, 32E - 1, R - 3, X - 2, -X² - 4R + 4X]

[x - 1, 2x - y]

[xy + 2x - y - 2]

[B, C, c, d, e, f, G, h]

[e, f, c, d]

LazardProjection([x² + y² + z² - 1], z);

LazardProjection(%, y);

[x² + y² - 1]

[x - 1, x + 1]

```
testpoly := [randpoly([w, x, y, z], degree = 4)];
```

```
testpoly := [44w2yz + 29wx3 + 98wx2y - 23wxyz - 23wy + 87wz]
```

Using the option `printorder=true`, we can display the values for each stage of the tiebreaking process (lowest degree, lowest total degree among relevant terms, lowest number of relevant terms, highest alphabetically):

```
brownord := BrownHeuristic(testpoly, printorder = true);
```

```
[[x, 3, 4, 3, 2], [w, 2, 4, 6, 1], [y, 1, 4, 4, 3], [z, 1, 4, 3, 4]]
brownord := [x, w, y, z]
```

The additional arguments of `GetCADOOrdering` allow the user to ‘set aside’ certain variables to be eliminated before or after the other variables present (default options are `position=bottom` and `outputorder=increasing`).

```
GetCADOOrdering(testpoly);
```

```
GetCADOOrdering(testpoly, [x, y]);
```

```
GetCADOOrdering(testpoly, [x, y], position = top);
```

```
GetCADOOrdering(testpoly, [x, y], position = top, outputorder = decreasing);
```

```
[x, w, y, z]
```

```
[x, y, w, z]
```

```
[w, z, x, y]
```

```
[y, x, z, w]
```

When `LazardProjection` is called without specifying a variable, the next variable to project will be chosen using `BrownHeuristic`, and the variable chosen will be displayed.

```
LazardProjection(testpoly);
```

```
LazardProjection(%);
```

```
LazardProjection(%);
```

Projecting with respect to z:

```
[w, 29x3 + 98x2y - 23y, 44wy - 23xy + 87]
```

Projecting with respect to w:

```
[y, 23xy - 87, 29x3 + 98x2y - 23y]
```

Projecting with respect to y:

```
[x, 98x2 - 23, 23x4 + 294x2 - 69]
```

Note that this is recalculated each time, and differs from the order supplied by calling `BrownHeuristic` on the initial set of polynomials:

```
LazardProjection(testpoly, op(-1, brownord));
LazardProjection(%, op(-2, brownord));
LazardProjection(%, op(-3, brownord));
```

$$\begin{aligned} &[w, 29x^3 + 98x^2y - 23y, 44wy - 23xy + 87] \\ &[w, x, 44w - 23x, 98x^2 - 23, 44wx^2 - 23x^4 - 294x^2 + 69] \\ &[x, 98x^2 - 23, 23x^4 + 294x^2 - 69] \end{aligned}$$

By default, `RecursiveLazard` will also project using this ordering, returning each projection polynomial set in the order from least to most variables.

```
RecursiveLazard(testpoly);
RecursiveLazard(testpoly, [x, y]);
RecursiveLazard(testpoly, [y, z]);
```

$$\begin{aligned} &[[x, 98x^2 - 23, 23x^4 + 294x^2 - 69], [w, x, 44w - 23x, 98x^2 - 23, 44wx^3 - 23x^4 - 294x^2 + 69], \\ &[w, 29x^3 + 98yx^2 - 23y, 44wy - 23xy + 87], [44w^2yz + 29wx^3 + 98wx^2y - 23wxyz - 23wy + 87zw]] \\ &[[x, 98x^2 - 23, 23x^4 + 294x^2 - 69], [y, 23xy - 87, 29x^3 + 98yx^2 - 23y], \\ &[w, 29x^3 + 98yx^2 - 23y, 44wy - 23xy + 87], [44w^2yz + 29wx^3 + 98wx^2y - 23wxyz - 23wy + 87zw]] \\ &[[y, 279841y^4 - 17060526y^2 - 19096587, 21647416y^2 + 230090031, 3764768y^2 - 522261], \\ &[y, z, 23y - 87z, 1411372z^3y^3 + 5080516z^2y^4 - 102363156z^2y^2 - 300473292zy^3 + 86589664y^4 - \\ &171869283z^2 + 90873414zy - 12012003y^2], [y, z, 29x^3 + 98yx^2 - 23yzx - 23y + 87z], \\ &[44w^2yz + 29wx^3 + 98wx^2y - 23wxyz - 23wy + 87zw]] \end{aligned}$$

In addition to inheriting the ordering options from `GetCADOOrdering`, `RecursiveLazard` also contains the option to cease projection once only the supplied variables are left (`stopatvars=true`) and the option to return only the final set of projection polynomials produced (`fulllist=false`).

```
RecursiveLazard(testpoly, [x, y], projorder = increasing, varposition = bottom,
fulllist = false, stopatvars = true);
```

$$[y, 23xy - 87, 29x^3 + 98x^2y - 23y]$$

We are also able to ‘project’ using `RegularChains` by producing a CCD and returning the level k polynomials present within. We are also able to specify an ordering and export the CCD produced.

```

GetProjPolysFromRC(testpoly);
GetProjPolysFromRC(testpoly, [z, y], projorder = increasing,
varposition = bottom, CD = CDOUT);
CDOUT;

[[x, 98x2 - 23, 23x4 + 294x2 - 69], [w, 44w - 23x, 44x3w - 23x4 - 294x2 + 69], [44wy -
23xy + 87], [44yzw + 29x3 + 98yx2 - 23yzx - 23y + 87z]]

[[z, 23z2 + 392, 12167z2 + 2979102, 36501z4 + 8937306z2 - 12167], [y, 5080516y4z2 +
1411372y3z3 + 86589664y4 - 300473292y3z - 102363156y2z2 - 12012003y2 +
90873414yz - 171869283z2], [29x3 + 98yx2 - 23yzx - 23y + 87z], [w, 44yzw + 29x3 +
98yx2 - 23yzx - 23y + 87z]]

c_c_d

```

These are both combined into the command **GetProjPolys**, allowing the user to produce projection polynomials using either method using the option **method=PL** or **method=RC**, defaulting to PL if no argument is specified. Generally, PL produces more polynomials, but RC can take longer to output, due to first producing a CCD and then extracting the polynomials.

```

print("PL - Lazard projection"); GetProjPolys(testpoly, method = PL);
print("RC - from CCT"); GetProjPolys(testpoly, method = RC);

"PL - Lazard projection"
[[x, 98x2 - 23, 23x4 + 294x2 - 69], [w, x, 44w - 23x, 98x2 - 23, 44w x3 - 23x4 - 294x2 + 69],
[w, 29x3 + 98y x2 - 23y, 44wy - 23xy + 87], [44w2yz + 29w x3 + 98w x2y - 23wxyz -
23wy + 87zw]]

"RC - from CCT"
[[x, 98x2 - 23, 23x4 + 294x2 - 69], [w, 44w - 23x, 44x3w - 23x4 - 294x2 + 69],
[44wy - 23xy + 87], [44yzw + 29x3 + 98yx2 - 23yzx - 23y + 87z]]

GetProjPolys(testpoly);
GetProjPolys(testpoly, method = othermethod);

[[x, 98x2 - 23, 23x4 + 294x2 - 69], [w, x, 44w - 23x, 98x2 - 23, 44w x3 - 23x4 - 294x2 + 69],
[w, 29x3 + 98y x2 - 23y, 44wy - 23xy + 87], [44w2yz + 29w x3 + 98w x2y - 23wxyz -
23wy + 87zw]]

```

Error, (in GetProjPolys) Incorrect method: options are 'PL' or 'RC'

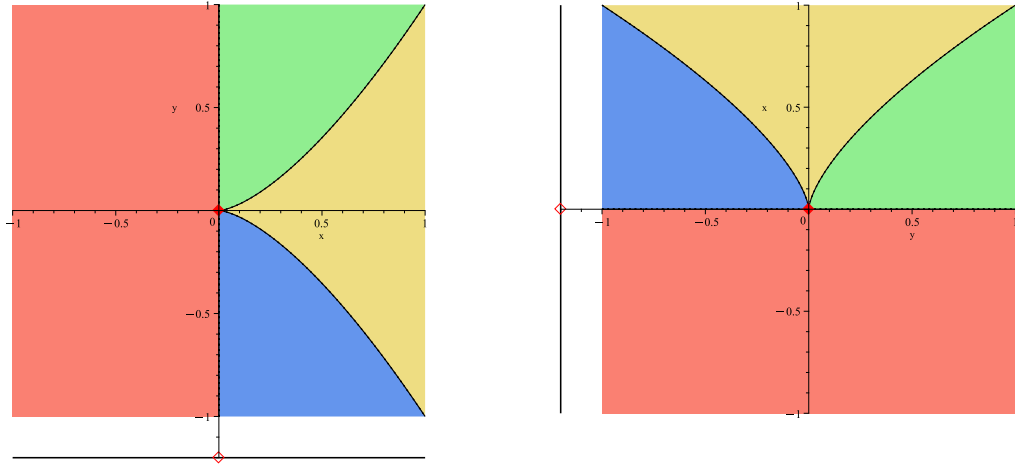


Figure 6-2: CAD graphs of $x^3 - y^2$ using PL and RC respectively. Both have the projection ordering $x \prec y$, but different display orderings, with the 1D CAD below.

Finally, we present the titular command **GraphCAD**, which produces a CAD graph directly from a set of polynomials and allows the user to specify both the projection and display order.

```
GraphCAD([x^3 - y^2], varorder = [x, y], displayorder = [x, y], CADmethod = PL);
GraphCAD([x^3 - y^2], varorder = [y, x], displayorder = [x, y], CADmethod = PL);
GraphCAD([x^3 - y^2], varorder = [x, y], displayorder = [y, x], CADmethod = PL);
GraphCAD([x^3 - y^2], varorder = [y, x], displayorder = [y, x], CADmethod = PL);
```

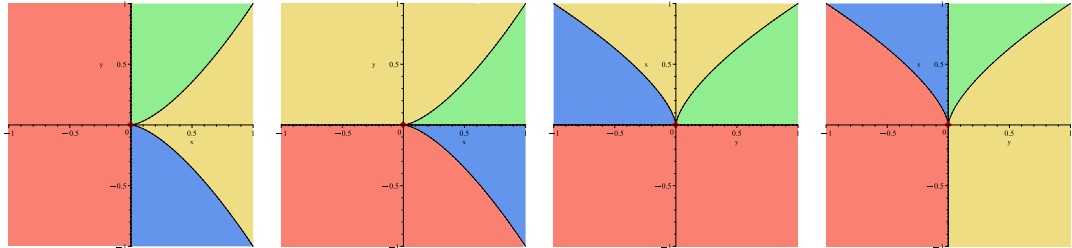


Figure 6-3: CAD graphs of $x^3 - y^2$ showing various combinations of projection and display order.

GraphCAD has additional options for displaying the 1D CAD below (`onedplot=true`), sample points (`samplepoints=true`), and only open cells (`opencad=true`):

```
GraphCAD([x^3 - y^2], varorder = [x, y], displayorder = [x, y], CADmethod = PL,
onedplot = true, samplepoints = true);
GraphCAD([x^3 - y^2], varorder = [x, y], displayorder = [x, y], CADmethod = PL,
opencad = true);
```

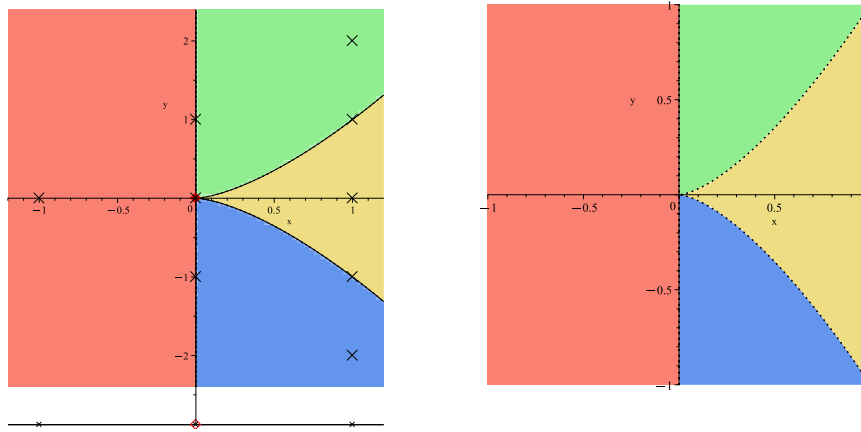


Figure 6-4: CAD graphs of $x^3 - y^2$ showing the 1D CAD below and sample points (left), and only open cells (right).

If the input polynomials are of higher dimension, **GraphCAD** will first obtain the 2D projection polynomials and construct the graph of those polynomials.

```
R1 := randpoly([x, y, z], degree=3, coeffs=rand(-5..5));
```

$$R1 := -2y^2x + 5y^3 - 3xz + 5y^2 - 4y$$

```
GraphCAD([R1], varorder = [op(indets(R1))], displayorder = [op(indets(R1))],
```

```
CADmethod = PL, xrange = -1..1, yrange = -1..1);
```

Warning, Not all variables supplied appear in polynomials.

Warning, Input polynomials have more than two variables, {x,y,z}

Warning, Using 2D projection polynomials, [x,y,2*x*y-5*y^2-5*y+4]

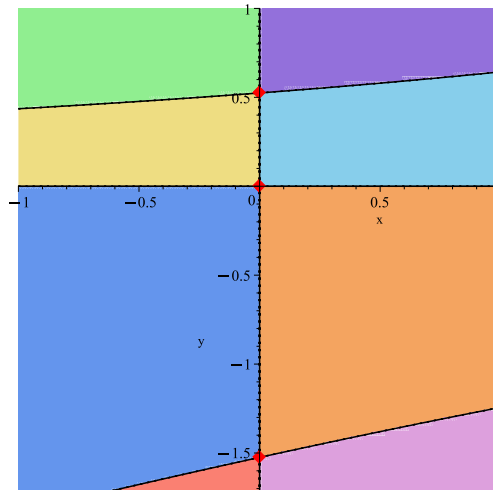


Figure 6-5: The 2D CAD associated to the 2D projection polynomials of $R1$.

Chapter 7

Future Work

In this thesis we have explored several avenues for improving CAD, such as special cases and features which can reduce the complexity of a problem, and additional implementations and communication tools which expand the reach of CAD to a wider user base. Further expansions to CAD and these ideas are given below.

7.1 Further Focus on Regular Chains

The incremental regular chains CAD algorithm (RC-CAD, see Section 2.9) is a newer alternative to the classical projection and lifting CAD algorithm (PL-CAD), and offers many advantages. It avoids nullification issues and typically produces fewer cells by avoiding redundant computation and can take advantage of multiple equational and inequational constraints present in the input polynomials, in contrast to PL-CAD, which can only exploit one EC per level. There are, however, some drawbacks to this method, which warrant further research and work:

1. **EC handling:** Is the saving made by exploiting a single EC with RC-CAD as powerful as one with PL-CAD? For PL-CAD, leveraging a single EC reduces the double-exponential complexity by one level, effectively lowering the problem dimension. For RC-CAD, leveraging an EC (or inequation) avoids the computation of certain cases, and while the benefits are outlined in Section 2.9.5, the actual savings in practice are less clear, especially in comparison to those of RC-CAD.
2. **Complexity analysis:** The experiments undertaken in Chapter 5 and earlier studies suggest RC-CAD is closely competitive with PL-CAD and generally produces fewer cells. Despite this, no theoretical complexity bounds have been determined. Doing so would help identify areas in which RC-CAD outperforms

PL-CAD, potentially allowing for a meta-algorithm to determine which method to use for a given input.

3. **Implementation input format:** Work has already been undertaken in Chapters 5 and 6 to handle the `RegularChains` implementation of RC-CAD’s specific requirements and connect it with other tools and implementations, but enabling direct handling of Tarski formulae or non-DNF formats would further increase usability and efficiency.
4. **Incremental-focused ordering:** Due to PL-CAD’s incremental nature, it cannot leverage levelwise variable ordering heuristics such as `gsotd` or `gmods`. However, heuristics which prioritise exploiting ECs and inequations, in a similar vein to Tonks, would make a considerable difference in reducing complexity. Additionally, tools for handling input polynomials, such as the use of Gröbner basis preprocessing for known ECs, and bespoke heuristics to order polynomials in addition to variables, would likely reap considerable benefits. For example, as the tree refinement involves polynomial GCDs, is there an optimal polynomial with which to begin tree refinement?

7.2 Strategies for Effective Use of CAD

CAD can be used to solve a wide range of problems, but the way these problems are presented and interpreted can significantly impact performance. Ways to address this are given below.

7.2.1 Formula and Ordering Considerations

There are opportunities to expand on the ideas in Chapter 3, relating to input formula consideration, feature identification and heuristic guidance:

1. **Implementing block orderings:** A step towards the implementation of block ordering (Section 2.8.3) has been made via the command `GetCADOrdering` (see Chapter 6), allowing the user to specify groups of variables to eliminate first or last. This can be extended to support block orderings, allowing the user to fix certain variables and their relative order, and obtaining a heuristic ordering only for the variables within blocks.
2. **Symmetry considerations:** By identifying further examples containing the $\mathbb{Z}_2$ symmetry displayed in the Solotareff problem (Chapter 3), one could generalise a method and variable ordering heuristic to facilitate exploiting such symmetry, which could then be extended to larger symmetry groups.

7.2.2 Other Improvements and Heuristics

Effective quantification (Section 3.2.6) touches on the idea that one need not accept the problem as given, and alternative formulations may reduce complexity. A way to determine if alternative formulations exist, or even if QE and CAD are feasible, would be of significant benefit, especially to users concerned by high complexity.

QE and CAD are general-purpose tools: able to handle any input in the correct format, but with doubly-exponential time. This extreme worst case and uncertainty may deter some users, and approximate time estimations can help address this. While worst-case bounds are unlikely to be helpful, approximations determined via experimentation may be of more benefit. Similarly, heuristics based on the degree, dimension, or polynomial count of the input, or on level 1 cell count, could be of great use in estimating a projected timeframe or for predicting infeasibility.

7.3 CylindricalAlgebraicDecomposition Improvements

Some possible directions for expanding the package include:

- **Improved Output:** Enhancing `liftingPoint` by evaluating the input polynomials at the top-level sample points, providing more information to users and streamlining `positivePoint`. This would also simplify manual sign-checking of cells and future related commands.
- **Extension to Full CAD:** Expanding from open CAD to a full CAD with cells of every dimension. This would require the implementation of real polynomial division and a way to symbolically store and retrieve real roots, but would enable the following two applications:
- **Quantifier Elimination:** A full CAD could be used to answer any non-linear arithmetic quantified questions through quantifier elimination (given sufficient resources).
- **Connectivity Analysis of Algebraic Varieties:** By combining a full CAD with an adjacency algorithm, such as the one described in [Str17], one could determine the number of connected components in algebraic varieties, offering new insights into their topological properties.

These ideas represent potential avenues for further research and development, with an aim to enhance understanding and capability in computational algebraic geometry.

7.4 Future Benchmarking Studies

One obvious piece of future work stemming from these benchmarking experiments is to look further at specific examples in which one algorithm did particularly well or poorly, and attempt to determine what features of these examples cause this behaviour. For example, PLCAD timed out on two examples in Section 5.6.3, but was otherwise only proportionally slower than RCCAD, and determining any features which may be the cause would be worthwhile.

One limitation of these experiments was the lack of a large real-world dataset containing multiple equational constraints and reasonable complexity in terms of degree, dimension, and polynomials. The dataset used in these experiments has groundings in industry, but ECs were added artificially. This dataset is known to be unbalanced in terms of variable composition, which is not an issue for our purposes, but the augmented dataset that corrects this contains only polynomials and therefore could not be used to test the effectiveness of ECs.

A worthwhile piece of future work would be to collate a dataset containing real-world examples with multiple ECs or features exploitable with TTICAD, or synthesise examples that are representative of real-world problems. A larger and more varied dataset would allow for clearer identification of features that enable one CAD construction method to outperform the other, potential leading to a heuristic for deciding which method is best suited to a given input.

7.5 Future Improvements for GraphCAD

There are many avenues for future development of the CAD graphing tool. A natural next step is to extend its functionality beyond two dimensions. Currently, the tool projects any higher-dimensional input into two dimensions and visualises the associated CAD, discarding any higher dimensional information. A worthwhile improvement would be to annotate each sample point of the 2D CAD with information on the number of cells lying above it in higher dimensions. However, this has the tradeoff of requiring the construction of a higher-dimensional CAD, which the current method avoids, but would bring with it the ability to accept higher-dimensional CADs as input. As the sample points of a CAD are themselves cylindrical, lower-dimensional ones could also be derived from higher-dimensional ones, meaning a method for extracting sample points and projection polynomials would allow this to be more easily generalised.

Similarly, the visualisation could be generalised to support CADs of one and three dimensions. Although 1D CADs can already be displayed as an extension of 2D CAD

visualisation, a way to store CADs of one to three dimensions and display them separately or simultaneously would be very worthwhile as a tool for communication and understanding of both CAD structure and projection.

Another important area of development is truth-invariant visualisation. At present, the tool produces a sign-invariant CAD using the polynomials present in the input formula, discarding extra information and any potential improvements that could be obtained by switching to truth-invariance. An important first step would be using lists of polynomials with constraints to determine the truth values of sample points, enabling the use of a truth-invariant colour-scheme. Full support for interpreting input formulae and restricted projections would then allow for a way to demonstrate the benefit and power of such simplifications, both visually and through improved performance.

Considerable groundwork has already been laid between this tool and those of Chapter 5, in terms of handling and translating more complicated inputs, and implementing various features directly, such as the Lazard projection and Brown heuristic. The next step is to directly interface with formulae and constraints. Similarly, a way to independently call other variable ordering heuristics (including those mentioned in Section 2.7 and the Tonks and `dsum` heuristics from Section 5.2.1) would greatly help to compare their effects, aiding in communication and research. Additionally, implementing other historical and recent projection operators would further extend the scope for communication and comparison.

There is also scope to improve user experience and efficiency through additional colour schemes. The current scheme colours 2-cells using various colours, 1-cells in black and 0-cells as red diamonds. Other useful alternatives include:

- **ByDimension:** Cells are coloured according to dimension, colouring 3-cells pale yellow, 2-cells pale red, 1-cells in blue and 0-cells in green.
- **ColourScale:** Cells are coloured using multiple shades of a given base colour.
- **Truth:** Cells are coloured varying shades of red and green to highlight truth value (using lighter shades for higher-dimensional cells).

These can also be used to improve the package’s overall efficiency. Currently, each 1-cell is drawn individually from the CAD output, so every line segment corresponds to a separate cell. A simpler and more efficient approach would be to draw each polynomial curve in full and then overlay the 0-cells. Similarly, 2-cell colouring could be reduced to a single colour for the plane. Simplifying a 2D CAD visualisation to polynomials, points and a background colour would likely yield significant improvements in both performance and memory.

Bibliography

- [Ach56] N.I. Achieser. *Theory of Approximation*. Frederick Ungar Publishing Co., New York, 1956.
- [ACM84] Dennis Arnon, George Collins, and Scott McCallum. Cylindrical Algebraic Decomposition I: The Basic Algorithm. *SIAM J. Comput.*, 13:865–877, 11 1984.
- [ADD25] Benjamin Aymard, Andrea Delahaye, and Audric Drogoul. An original classification of obscuration-free telescopes designs unfolded in two dimensions. *ArXiv pre-print*, 2025. Available at: <https://arxiv.org/abs/2501.04748>.
- [AP10] Behzad Akbarpour and Lawrence Charles Paulson. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning*, 44(3):175–205, Mar 2010.
- [Arn81] Dennis Soule Arnon. *Algorithms for the geometry of semi-algebraic sets*. PhD thesis, The University of Wisconsin - Madison, 1981. AAI8120294.
- [BCD⁺14] Russell Bradford, Changbo Chen, James H. Davenport, Matthew England, Marc Moreno Maza, and David Wilson. Truth Table Invariant Cylindrical Algebraic Decomposition by Regular Chains. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 44–58. Springer International Publishing, 2014.
- [BCR98] Jacek Bochnak, Michel Coste, and Marie-Françoise Roy. *Real Algebraic Geometry*. Springer Berlin, Heidelberg, 1998.
- [BD07] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, page 54–60, New York, NY, USA, 2007. Association for Computing Machinery.
- [BDE⁺13] Russell Bradford, James H. Davenport, Matthew England, Scott McCallum, and David Wilson. Cylindrical algebraic decompositions for boolean combinations. In *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, ISSAC '13, page 125–132, New York, NY, USA, 2013. Association for Computing Machinery.

- [BDE⁺16] Russell Bradford, James H. Davenport, Matthew England, Scott McCallum, and David Wilson. Truth table invariant cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 76:1–35, 2016.
- [BDE⁺17] Russell Bradford, James H. Davenport, Matthew England, Hassan Errami, Vladimir Gerdt, Dima Grigoriev, Charles Hoyt, Marek Košta, Ovidiu Radulescu, Thomas Sturm, and Andreas Weber. A case study on the parametric occurrence of multiple steady states. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation, ISSAC '17*, page 45–52, New York, NY, USA, 2017. Association for Computing Machinery.
- [BDG17] M. Brain, J.H Davenport, and A. Griggio. Benchmarking Solvers, SAT-style. In *Proceedings of the 2nd International Workshop on Satisfiability Checking and Symbolic Computation*, volume 1974 of *CEUR Workshop Proceedings*, 2017.
- [BENW06] Christopher W. Brown, M’hammed El Kahoui, Dominik Novotni, and Andreas Weber. Algorithmic methods for investigating equilibria in epidemic modeling. *Journal of Symbolic Computation*, 41(11):1157–1173, 2006. Special Issue on the Occasion of Volker Weispfenning’s 60th Birthday.
- [BM20] Christopher W. Brown and Scott McCallum. Enhancements to Lazard’s Method for Cylindrical Algebraic Decomposition. In François Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 129–149. Springer International Publishing, 2020.
- [Bro] Christopher W Brown. Q E P C A D - Quantifier Elimination by Partial Cylindrical Algebraic Decomposition. Available at: <https://www.usna.edu/CS/qepcadweb/B/QEPCAD.html>. Accessed: 2025-06-27.
- [Bro04] Christopher W. Brown. Companion to the Tutorial Cylindrical Algebraic Decomposition Presented at ISSAC 2004. Available at: <https://www.usna.edu/Users/cs/wcbrown/research/ISSAC04/handout.pdf>, 2004. Accessed: 2025-11-26.
- [CC18] A. E. Charalampakis and I. Chatziannellis. Analytical solutions for the minimum weight design of trusses by cylindrical algebraic decomposition. *Archive of Applied Mechanics*, 88(1):39–49, 2018.
- [CH91] George E. Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991.
- [CKJ⁺15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368. Springer International Publishing, 2015.

- [CMM14a] Changbo Chen and Marc Moreno Maza. An Incremental Algorithm for Computing Cylindrical Algebraic Decompositions. In Ruyong Feng, Wen-shin Lee, and Yosuke Sato, editors, *Computer Mathematics*. Springer Berlin Heidelberg, 2014. Presented at ASCM 2012. Also available as arXiv:1210.5543.
- [CMM14b] Changbo Chen and Marc Moreno Maza. Cylindrical Algebraic Decomposition in the RegularChains Library. In Hoon Hong and Chee Yap, editors, *Mathematical Software – ICMS 2014*, pages 425–433. Springer Berlin Heidelberg, 2014.
- [CMM14c] Changbo Chen and Marc Moreno Maza. Quantifier Elimination by Cylindrical Algebraic Decomposition Based on Regular Chains. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC ’14, page 91–98, New York, NY, USA, 2014. Association for Computing Machinery.
- [CMMXY09] Changbo Chen, Marc Moreno Maza, Bican Xia, and Lu Yang. Computing Cylindrical Algebraic Decomposition via Triangular Decomposition. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC ’09, page 95–102, New York, NY, USA, 2009. Association for Computing Machinery.
- [Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, pages 134–183. Springer Berlin Heidelberg, 1975.
- [Col96] George E. Collins. Application of Quantifier Elimination to Solotareff’s Approximation Problem. In Rolf Jeltsch and Mohamed Mansour, editors, *Stability Theory*, pages 181–190, Basel, 1996. Birkhäuser Basel.
- [Col98] George E. Collins. Quantifier Elimination by Cylindrical Algebraic Decomposition — Twenty Years of Progress. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 8–23. Springer Vienna, 1998.
- [CSEDS20] Jose Capco, Mohab Safey El Din, and Josef Schicho. Robots, computer algebra and eight connected components. In *ISSAC’20: Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*, ISSAC’20: Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation, pages 62–69, Kalamata / Virtual, Greece, July 2020. ACM.
- [Dav] J. H. Davenport. Computer Algebra. Available at: <http://staff.bath.ac.uk/masjhd/JHD-CA.pdf>. Accessed: 2023-10-27.
- [DBEW12] James H. Davenport, Russell Bradford, Matthew England, and David Wilson. Program verification in the presence of complex numbers, functions with branch cuts etc. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 83–88, 2012.
- [DH88] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1):29–35, 1988.

- [DNSU23] James Harold Davenport, Akshar Sajive Nair, Gregory Kumar Sankaran, and Ali Kemal Uncu. Lazard-style CAD and Equational Constraints. In *Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation*, ISSAC '23, page 218–226, New York, NY, USA, 2023. Association for Computing Machinery.
- [dRE22] Tereso del Río and Matthew England. New Heuristic to Choose a Cylindrical Algebraic Decomposition Variable Ordering Motivated by Complexity Analysis. In François Boulrier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 300–317. Springer International Publishing, 2022.
- [DSS04] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient projection orders for CAD. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, page 111–118, New York, NY, USA, 2004. Association for Computing Machinery.
- [DSW98] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. A New Approach for Automatic Theorem Proving in Real Geometry. *Journal of Automated Reasoning*, 21(3):357–380, Dec 1998.
- [EBD20] Matthew England, Russell Bradford, and James H. Davenport. Cylindrical algebraic decomposition with equational constraints. *Journal of Symbolic Computation*, 100:38–71, Sep 2020.
- [EF19] Matthew England and Dorian Florescu. Comparing Machine Learning Models to Choose the Variable Ordering for Cylindrical Algebraic Decomposition. In Cezary Kaliszyk, Edwin Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 93–108. Springer International Publishing, 2019.
- [EH16] Mădălina Erăscu and Hoon Hong. Real quantifier elimination for the synthesis of optimal numerical algorithms (Case study: Square root computation). *Journal of Symbolic Computation*, 75:110–126, 2016. Special issue on the conference ISSAC 2014: Symbolic computation and computer algebra.
- [Eng24] Matthew England. Real Quantifier Elimination Technology and Optimisations via Machine Learning. Available at: <https://matthewengland.coventry.domains/Conferences/2024-03-CAGMeeting.pdf>, March 2024. Invited talk, Computational Algebraic Geometry Research Network, University of Warwick.
- [FPM05] I.A. Fotiou, P.A. Parrilo, and M. Morari. Nonlinear parametric optimization using cylindrical algebraic decomposition. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3735–3740, 2005.
- [GMSY24] Jordy Lopez Garcia, Kelly Maluccio, Frank Sottile, and Thomas Yahl. Real solutions to systems of polynomial equations in Macaulay2. *Journal of Software for Algebra and Geometry*, 14(1):87–95, 2024.

- [GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at: <http://www2.macaulay2.com>. Accessed: 2025-11-18.
- [HEW⁺19] Zongyan Huang, Matthew England, David J. Wilson, James Bridge, James H. Davenport, and Lawrence C. Paulson. Using Machine Learning to Improve Cylindrical Algebraic Decomposition. *Mathematics in Computer Science*, 13(4):461–488, 2019.
- [HHP⁺23] J. Hester, B. Hitaj, G. Passmore, S. Owre, N. Shankar, and E. Yeh. An Augmented MetiTarski Dataset for Real Quantifier Elimination Using Machine Learning. In Catherine Dubois and Manfred Kerber, editors, *Proceedings CICM 2023*, volume 14101 of *Springer Lecture Notes in Computer Science*, pages 297–302, 2023.
- [Hon90] Hoon Hong. *Improvements in CAD-based quantifier elimination*. PhD thesis, The Ohio State University, 1990.
- [Hon92] Hoon Hong. Simple solution formula construction in cylindrical algebraic decomposition based quantifier elimination. In *Papers from the International Symposium on Symbolic and Algebraic Computation*, ISSAC '92, page 177–188, New York, NY, USA, 1992. Association for Computing Machinery.
- [Ioa17a] Nikolaos I. Ioakimidis. Application of quantifier elimination to inverse buckling problems. *Acta Mechanica*, 228(10):3709–3724, Oct 2017.
- [Ioa17b] Nikolaos I. Ioakimidis. Application of quantifier elimination to mixed-mode fracture criteria in crack problems. *Archive of Applied Mechanics*, 87(10):1567–1604, Oct 2017.
- [JD24] Rohit John and James Davenport. Exploring Alternative Machine Learning Models for Variable Ordering in Cylindrical Algebraic Decomposition. In Kevin Buzzard, Alicia Dickenstein, Bettina Eick, Anton Leykin, and Yue Ren, editors, *Mathematical Software — ICMS 2024*, volume 14749 of *Springer Lecture Notes in Computer Science*, pages 176–185. Springer, 2024.
- [JdM12] Dejan Jovanović and Leonardo de Moura. Solving Non-linear Arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 339–354. Springer Berlin Heidelberg, 2012.
- [Jir95] Mats Jirstrand. Cylindrical Algebraic Decomposition - an Introduction. Technical Report 1807, Linköping University, Department of Electrical Engineering, Automatic Control, 1995.
- [Laz94] D. Lazard. *An Improved Projection for Cylindrical Algebraic Decomposition*, volume 26 of *Springer Series in Computational Mathematics*, pages 467–476. Springer New York, 1994.

- [LdR25] Corin Lee, Tereso del Río, and Hamid Rahkooy. Cylindrical Algebraic Decomposition in Macaulay2. *ArXiv pre-print*, 2025. Available at: <https://arxiv.org/abs/2503.21731>.
- [Lee25a] Corin Lee. GraphCAD, in CADRepository - GitHub. Available at: <https://github.com/cel34-bath/CADRepository/tree/main/GraphCAD>, Nov 2025. Accessed: 2025-11-18.
- [Lee25b] Corin Lee. Macaulay2, in CADRepository - GitHub. Available at: <https://github.com/cel34-bath/CADRepository/tree/main/Macaulay2>, Jun 2025. Accessed: 2025-11-18.
- [Mapa] Maplesoft. Overview of the QuantifierElimination package. Available at: <https://www.maplesoft.com/support/help/Maple/view.aspx?path=QuantifierElimination>. Accessed: 2025-07-15.
- [Mapb] Maplesoft. Overview of the RegularChains Package. Available at: <https://www.maplesoft.com/support/help/Maple/view.aspx?path=RegularChains>. Accessed: 2025-07-15.
- [McC84] S. McCallum. *An Improved Projection Operation for Cylindrical Algebraic Decomposition*. PhD thesis, University of Wisconsin–Madison, 1984.
- [McC85] Scott McCallum. An Improved Projection Operation for Cylindrical Algebraic Decomposition. Technical Report 578, University of Wisconsin-Madison Department of Computer Sciences, 1985.
- [McC88] Scott McCallum. An improved projection operation for cylindrical algebraic decomposition of three-dimensional space. *Journal of Symbolic Computation*, 5(1):141–161, 1988.
- [McC98] Scott McCallum. An Improved Projection Operation for Cylindrical Algebraic Decomposition. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 242–268. Springer Vienna, 1998.
- [McC99] Scott McCallum. On Projection in CAD-Based Quantifier Elimination with Equational Constraint. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, ISSAC '99, page 145–149, New York, NY, USA, 1999. Association for Computing Machinery.
- [McC01] Scott McCallum. On Propagation of Equational Constraints in CAD-Based Quantifier Elimination. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, ISSAC '01, page 223–231, New York, NY, USA, 2001. Association for Computing Machinery.
- [MDE18] Casey B. Mulligan, James H. Davenport, and Matthew England. TheoryGuru: A Mathematica Package to Apply Quantifier Elimination Technology to Economics. In James H. Davenport, Manuel Kauers, George Labahn, and Josef

- Urban, editors, *Mathematical Software – ICMS 2018*, pages 369–378. Springer International Publishing, 2018.
- [MH16] Scott McCallum and Hoon Hong. On using Lazard’s projection in CAD construction. *Journal of Symbolic Computation*, 72:65–81, 2016.
- [MM82] Ernst W Mayr and Albert R Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Advances in Mathematics*, 46(3):305–329, 1982.
- [MMC⁺12] Montserrat Manubens, Guillaume Moroz, Damien Chablat, Philippe Wenger, and Fabrice Rouillier. Cusp Points in the Parameter Space of Degenerate 3-RPR Planar Parallel Manipulators. *Journal of Mechanisms and Robotics*, 4(4):041003, 08 2012.
- [MPP19] Scott McCallum, Adam Parusiński, and Laurentiu Paunescu. Validity proof of Lazard’s method for CAD construction. *Journal of Symbolic Computation*, 92:52–69, 2019.
- [Nai21] Akshar Nair. *Curtains in Cylindrical Algebraic Decomposition*. PhD thesis, University of Bath, 2021.
- [NDS20] Akshar Nair, James Davenport, and Gregory Sankaran. Curtains in CAD: Why Are They a Problem and How Do We Fix Them? In *Mathematical Software – ICMS 2020 - 7th International Conference, Proceedings*, Lecture Notes in Computer Science, pages 17–26. Springer, Singapore, July 2020. 7th International Congress on Mathematical Software, ICMS 2020 ; Conference date: 13-07-2020 Through 16-07-2020.
- [Pau12] Lawrence C. Paulson. MetiTarski: Past and Future. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 1–10. Springer Berlin Heidelberg, 2012.
- [PdEC24] Lynn Pickering, Tereso del Río Almajano, Matthew England, and Kelly Cohen. Explainable AI Insights for Symbolic Computation: A case study on selecting the variable ordering for cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 123:102276, 2024.
- [Res] Wolfram Research. Wolfram Mathematica: Modern Technical Computing. Available at: <http://www.wolfram.com/mathematica/>. Accessed: 2022-10-21.
- [RS21] Gergely Röst and AmirHosein Sadeghimanesh. Exotic Bifurcations in Three Connected Populations with Allee Effect. *International Journal of Bifurcation and Chaos*, 31(13):2150202, 2021.
- [RV19] Heinz Joachim Rack and Robert Vajda. Explicit algebraic solution of Zolotarev’s First Problem for low-degree polynomials. *J. Numer. Anal. Approx. Theory*, 48(2):175–201, Dec. 2019.

- [Sei54] A. Seidenberg. A New Decision Method for Elementary Algebra. *Annals of Mathematics*, 60(2):365–374, 1954.
- [SS83] Jacob T. Schwartz and Micha Sharir. On the “piano movers” problem. II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics*, 4(3):298–351, 1983.
- [ST11] Thomas Sturm and Ashish Tiwari. Verification and synthesis using real quantifier elimination. In *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation*, ISSAC ’11, page 329–336, New York, NY, USA, 2011. Association for Computing Machinery.
- [Str00] Adam Strzeboński. Solving Systems of Strict Polynomial Inequalities. *Journal of Symbolic Computation*, 29(3):471–480, 2000.
- [Str17] Adam Strzeboński. CAD Adjacency Computation Using Validated Numerics. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC ’17, page 413–420, New York, NY, USA, 2017. Association for Computing Machinery.
- [Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry: Prepared for Publication with the Assistance of J.C.C. McKinsey*. RAND Corporation, Santa Monica, CA, 1951.
- [Ton21] Zak Tonks. *Poly-algorithmic Techniques in Real Quantifier Elimination*. PhD thesis, University of Bath, 2021.
- [VEB18] Fernando Vale-Enriquez and Christopher W. Brown. Polynomial Constraints and Unsat Cores in Tarski. In James H. Davenport, Manuel Kauers, George Labahn, and Josef Urban, editors, *Mathematical Software – ICMS 2018*, pages 466–474. Springer International Publishing, 2018.
- [Wan98] D. Wang. Decomposing polynomial systems into simple systems. *Journal of Symbolic Computation*, 25(3):295–314, 1998.
- [Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1):3–27, 1988.
- [Wil14] David Wilson. *Advances in Cylindrical Algebraic Decomposition*. PhD thesis, University of Bath, 2014.
- [WMTA16] Yumi Wada, Takuya Matsuzaki, Akira Terui, and Noriko H. Arai. An Automated Deduction and Its Implementation for Solving Problem of Sequence at University Entrance Examination. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese, editors, *Mathematical Software – ICMS 2016*, pages 82–89. Springer International Publishing, 2016.
- [YA07] Hitoshi Yanami and Hirokazu Anai. SyNRAC: A Maple Toolbox for Solving Real Algebraic Constraints. *ACM Commun. Comput. Algebra*, 41(3):112–113, Sep 2007.